
SPE Runtime Management Library

Version 2.0

April 9, 2016

©Copyright International Business Machines Corporation, Sony Computer Entertainment Incorporated, Toshiba Corporation 2003, 2004, 2005, 2006

All Rights Reserved

Printed in the United States of America June 2006

The following are trademarks of International Business Machines Corporation in the United States, or other countries, or both.

IBM PowerPC

IBM Logo PowerPC Architecture

Other company, product, and service names may be trademarks or service marks of others. All information contained in this document is subject to change without notice. The products described in this document are NOT intended for use in applications such as implantation, life support, or other hazardous uses where malfunction could result in death, bodily injury, or catastrophic property damage. The information contained in this document does not affect or change IBM product specifications or warranties. Nothing in this document shall operate as an express or implied license or indemnity under the intellectual property rights of IBM or third parties. All information contained in this document was obtained in specific environments, and is presented as an illustration. The results obtained in other operating environments may vary.

THE INFORMATION CONTAINED IN THIS DOCUMENT IS PROVIDED ON AN "AS IS" BASIS. In no event will IBM be liable for damages arising directly or indirectly from any use of the information contained in this document.

IBM Systems and Technology Group
2070 Route 52, Bldg. 330
Hopewell Junction, NY 12533-6351

The IBM home page can be found at ibm.com

The IBM semiconductor solutions home page can be found at ibm.com/chips

June 15, 2006

Contents

1	Overview	1
1.1	Terminology	1
1.2	Usage Scenarios	1
2	Data Structure Documentation	5
2.1	addr64 Union Reference	5
2.2	fd_attr Struct Reference	6
2.3	image_handle Struct Reference	7
2.4	mfc_command_parameter_area Struct Reference	8
2.5	spe_context Struct Reference	9
2.6	spe_context_base_priv Struct Reference	11
2.7	spe_context_event_priv Struct Reference	15
2.8	spe_context_info Struct Reference	17
2.9	spe_event_data Union Reference	18
2.10	spe_event_unit Struct Reference	19
2.11	spe_gang_context Struct Reference	21
2.12	spe_gang_context_base_priv Struct Reference	22
2.13	spe_ld_info Struct Reference	23
2.14	spe_mfc_command_area Struct Reference	24
2.15	spe_mssync_area Struct Reference	26
2.16	spe_program_handle Struct Reference	26
2.17	spe_reg128 Struct Reference	28
2.18	spe_sig_notify_1_area Struct Reference	28
2.19	spe_sig_notify_2_area Struct Reference	29
2.20	spe_spu_control_area Struct Reference	30
2.21	spe_stop_info Struct Reference	32
3	File Documentation	35
3.1	accessors.c File Reference	35
3.2	cbea_map.h File Reference	39
3.3	create.c File Reference	40
3.4	create.h File Reference	47
3.5	design.txt File Reference	49
3.6	dma.c File Reference	49
3.7	dma.h File Reference	55
3.8	elf_loader.c File Reference	57
3.9	elf_loader.h File Reference	61
3.10	handler_utils.h File Reference	66
3.11	image.c File Reference	68
3.12	info.c File Reference	71
3.13	info.h File Reference	74
3.14	lib_builtin.c File Reference	77

3.15	lib_builtin.h File Reference	80
3.16	libspe2-types.h File Reference	82
3.17	libspe2.h File Reference	89
3.18	load.c File Reference	93
3.19	mbox.c File Reference	96
3.20	mbox.h File Reference	105
3.21	run.c File Reference	106
3.22	spe_event.c File Reference	111
3.23	spebase.h File Reference	121
3.24	speevent.h File Reference	158
Index		169

Chapter 1

Overview

The libspe2 functionality is split into 4 libraries:

- **libspe-base** This library provides the basic infrastructure to manage and use SPEs. The central data structure is a SPE context [spe_context](#). It contains all information necessary to manage an SPE, run code on it, communicate with it, and so on. To use the libspe-base library, the header file [spebase.h](#) has to be included and an application needs to link against **libspebase.a** or **libspebase.so**.
- **libspe-event** This is a convenience library for the handling of events generated by an SPE. It is based on libspe-base and epoll. Since the [spe_context](#) introduced by libspe-base contains the file descriptors to mailboxes etc, any other event handling mechanism could also be implemented based on libspe-base.

1.1 Terminology

- **main thread** usually the application main thread running on a PPE
- **SPE thread** a thread that uses SPEs. Execution starts on the PPE. Execution shifts between PPE and an SPE back and fro, e.g., PPE services system calls for SPE transparently

1.2 Usage Scenarios

1.2.1 Single-threaded sample

Note: In the new model, it is not necessary to have a main thread - the SPE thread can be the only application thread. It may run parts of its code on PPE and then start an SPE, e.g., for an accelerated function. The main thread is needed only if you want to use multiple SPEs concurrently. The following minimalistic sample illustrates the basic steps:

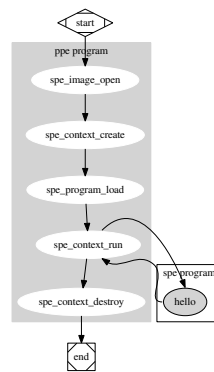


Figure 1.1: Simple program

Here is the same sample with some error checking:

1.2.2 Multi-threaded sample

This illustrates a threaded sample using the pthread library:

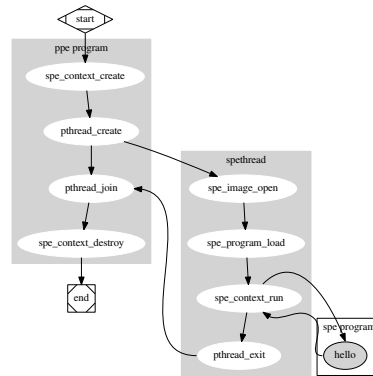


Figure 1.2: Simple pthread program

Here is the same sample with some error checking:

1.2.3 Problem state mapping samples

This illustrates accessing the MFC Local Store Address Register.

1.2.4 Event samples

This illustrates a sample using the event library. The event, which we receive is of course that the spu program has stopped, because otherwise we would not get there.

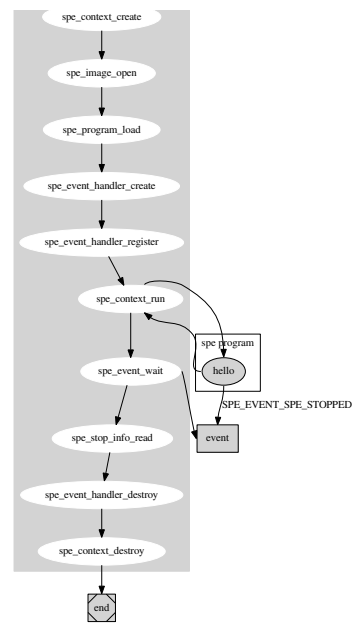


Figure 1.3: Simple event program

Events are more useful in multithreaded environments:

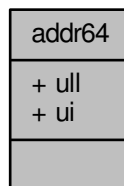
Chapter 2

Data Structure Documentation

2.1 addr64 Union Reference

```
#include <elf_loader.h>
```

Collaboration diagram for addr64:



Data Fields

- unsigned long long [ull](#)
- unsigned int [ui](#) [2]

2.1.1 Detailed Description

Definition at line 28 of file elf_loader.h.

2.1.2 Field Documentation

unsigned int [ui](#)[2]

Definition at line 31 of file elf_loader.h.

Referenced by `_base_spe_context_run()`.

unsigned long long [ull](#)

Definition at line 30 of file elf_loader.h.

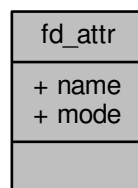
Referenced by `_base_spe_context_run()`.

The documentation for this union was generated from the following file:

- [elf_loader.h](#)

2.2 fd_attr Struct Reference

Collaboration diagram for fd_attr:



Data Fields

- `const char *` [name](#)
- `int` [mode](#)

2.2.1 Detailed Description

Definition at line 37 of file `create.c`.

2.2.2 Field Documentation

`int mode`

Definition at line 39 of file `create.c`.

Referenced by `_base_spe_open_if_closed()`.

`const char* name`

Definition at line 38 of file `create.c`.

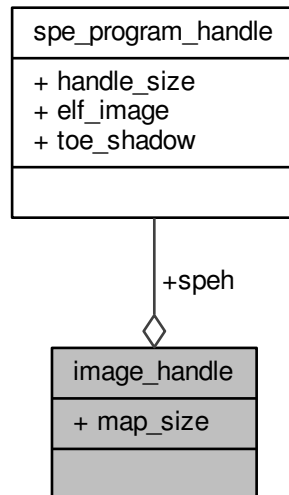
Referenced by `_base_spe_open_if_closed()`.

The documentation for this struct was generated from the following file:

- [create.c](#)

2.3 image_handle Struct Reference

Collaboration diagram for image_handle:



Data Fields

- [spe_program_handle_t speh](#)
- unsigned int [map_size](#)

2.3.1 Detailed Description

Definition at line 32 of file `image.c`.

2.3.2 Field Documentation

unsigned int `map_size`

Definition at line 34 of file `image.c`.

Referenced by `_base_spe_image_close()`, and `_base_spe_image_open()`.

`spe_program_handle_t speh`

Definition at line 33 of file `image.c`.

Referenced by `_base_spe_image_close()`, and `_base_spe_image_open()`.

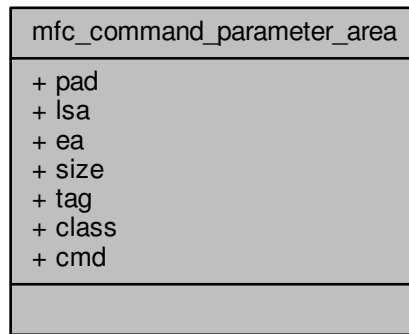
The documentation for this struct was generated from the following file:

- [image.c](#)

2.4 mfc_command_parameter_area Struct Reference

```
#include <dma.h>
```

Collaboration diagram for mfc_command_parameter_area:



Data Fields

- `uint32_t` [pad](#)
- `uint32_t` [lsa](#)
- `uint64_t` [ea](#)
- `uint16_t` [size](#)
- `uint16_t` [tag](#)
- `uint16_t` [class](#)
- `uint16_t` [cmd](#)

2.4.1 Detailed Description

Definition at line 27 of file `dma.h`.

2.4.2 Field Documentation

`uint16_t` class

Definition at line 33 of file `dma.h`.

`uint16_t` cmd

Definition at line 34 of file `dma.h`.

`uint64_t` ea

Definition at line 30 of file `dma.h`.

`uint32_t` lsa

Definition at line 29 of file `dma.h`.

uint32_t pad

Definition at line 28 of file dma.h.

uint16_t size

Definition at line 31 of file dma.h.

uint16_t tag

Definition at line 32 of file dma.h.

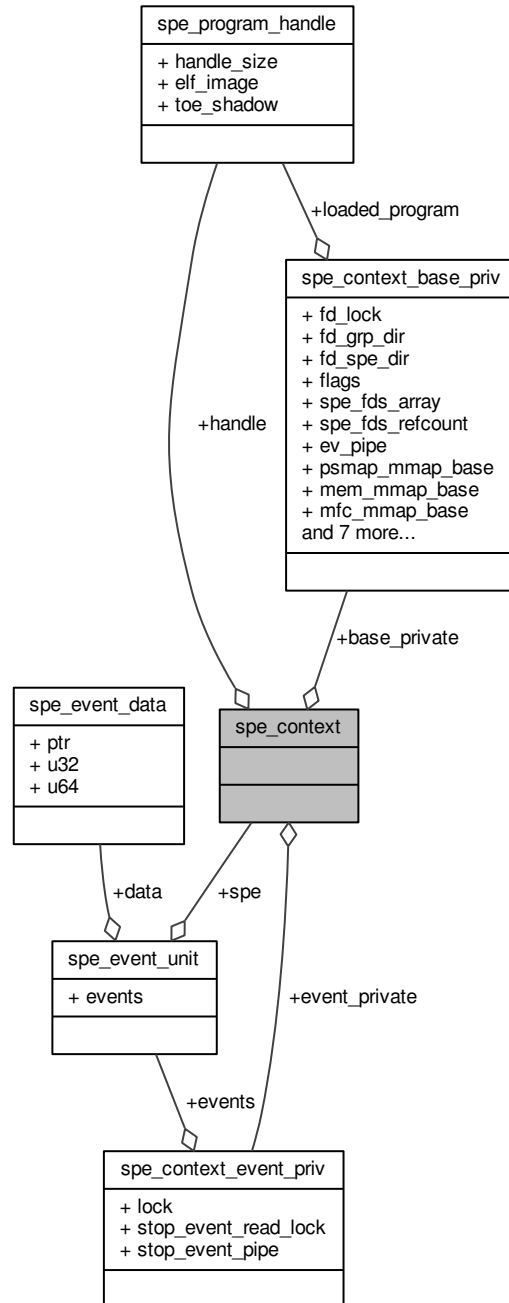
The documentation for this struct was generated from the following file:

- [dma.h](#)

2.5 spe_context Struct Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for `spe_context`:



Data Fields

- `spe_program_handle_t handle`
- `struct spe_context_base_priv * base_private`

- struct [spe_context_event_priv](#) * [event_private](#)

2.5.1 Detailed Description

SPE context The SPE context is one of the base data structures for the libspe2 implementation. It holds all persistent information about a "logical SPE" used by the application. This data structure should not be accessed directly, but the application uses a pointer to an SPE context as an identifier for the "logical SPE" it is dealing with through libspe2 API calls.

Definition at line 64 of file libspe2-types.h.

2.5.2 Field Documentation

struct [spe_context_base_priv](#)* [base_private](#)

Definition at line 76 of file libspe2-types.h.

Referenced by [_base_spe_spe_dir_get\(\)](#), [_base_spe_stop_event_source_get\(\)](#), [_base_spe_stop_event_target_get\(\)](#), [_base_spe_close_if_open\(\)](#), [_base_spe_context_create\(\)](#), [_base_spe_context_lock\(\)](#), [_base_spe_context_run\(\)](#), [_base_spe_context_unlock\(\)](#), [_base_spe_handle_library_callback\(\)](#), [_base_spe_in_mbox_status\(\)](#), [_base_spe_in_mbox_write\(\)](#), [_base_spe_ls_area_get\(\)](#), [_base_spe_mfcio_tag_status_read\(\)](#), [_base_spe_mssync_start\(\)](#), [_base_spe_mssync_status\(\)](#), [_base_spe_open_if_closed\(\)](#), [_base_spe_out_intr_mbox_status\(\)](#), [_base_spe_out_mbox_read\(\)](#), [_base_spe_out_mbox_status\(\)](#), [_base_spe_program_load\(\)](#), [_base_spe_program_load_complete\(\)](#), [_base_spe_ps_area_get\(\)](#), [_base_spe_signal_write\(\)](#), and [_event_spe_event_handler_register\(\)](#).

struct [spe_context_event_priv](#)* [event_private](#)

Definition at line 77 of file libspe2-types.h.

[spe_program_handle_t](#) [handle](#)

Definition at line 72 of file libspe2-types.h.

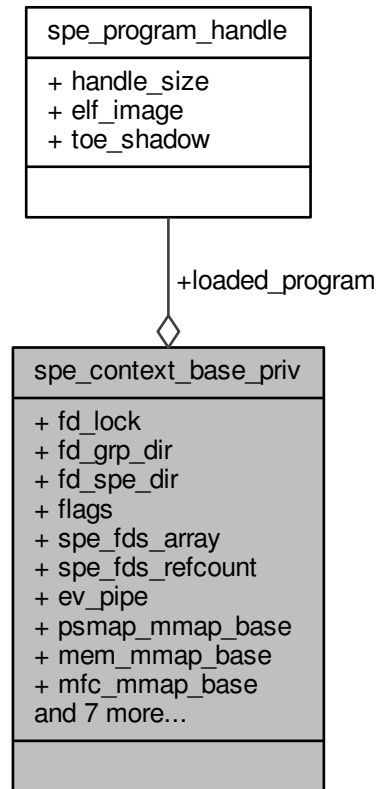
The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

2.6 [spe_context_base_priv](#) Struct Reference

```
#include <spebase.h>
```

Collaboration diagram for `spe_context_base_priv`:



Data Fields

- `pthread_mutex_t fd_lock` [NUM_MBOX_FDS]
- `int fd_grp_dir`
- `int fd_spe_dir`
- `unsigned int flags`
- `int spe_fds_array` [NUM_MBOX_FDS]
- `int spe_fds_refcount` [NUM_MBOX_FDS]
- `int ev_pipe` [2]
- `void * psmmap_mmap_base`
- `void * mem_mmap_base`
- `void * mfc_mmap_base`
- `void * mssync_mmap_base`
- `void * cntl_mmap_base`
- `void * signal1_mmap_base`
- `void * signal2_mmap_base`
- `int entry`

- [spe_program_handle_t](#) * [loaded_program](#)
- int [emulated_entry](#)
- int [active_tagmask](#)

2.6.1 Detailed Description

Definition at line 61 of file `spebase.h`.

2.6.2 Field Documentation

int active_tagmask

Definition at line 108 of file `spebase.h`.

Referenced by `_base_spe_mfcio_tag_status_read()`.

void* cntl_mmap_base

Definition at line 88 of file `spebase.h`.

Referenced by `_base_spe_context_create()`, `_base_spe_in_mbox_status()`, `_base_spe_out_intr_mbox_status()`, `_base_spe_out_mbox_status()`, and `_base_spe_ps_area_get()`.

int emulated_entry

Definition at line 103 of file `spebase.h`.

Referenced by `_base_spe_context_run()`, and `_base_spe_program_load()`.

int entry

Definition at line 93 of file `spebase.h`.

Referenced by `_base_spe_context_run()`, and `_base_spe_program_load()`.

int ev_pipe[2]

Definition at line 81 of file `spebase.h`.

Referenced by `__base_spe_stop_event_source_get()`, and `__base_spe_stop_event_target_get()`.

int fd_grp_dir

Definition at line 68 of file `spebase.h`.

pthread_mutex_t fd_lock[NUM_MBOX_FDS]

Definition at line 65 of file `spebase.h`.

Referenced by `_base_spe_context_create()`, `_base_spe_context_lock()`, and `_base_spe_context_unlock()`.

int fd_spe_dir

Definition at line 71 of file `spebase.h`.

Referenced by `__base_spe_spe_dir_get()`, `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_open_if_closed()`, and `_base_spe_program_load_complete()`.

unsigned int flags

Definition at line 74 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_handle_library_callback()`, `_base_spe_in_mbox_status()`, `_base_spe_in_mbox_write()`, `_base_spe_mfcio_tag_status_read()`, `_base_spe_mssync_start()`, `_base_spe_mssync_status()`, `_base_spe_out_intr_mbox_status()`, `_base_spe_out_mbox_read()`, `_base_spe_out_mbox_status()`, `_base_spe_program_load()`, `_base_spe_signal_write()`, and `_event_spe_event_handler_register()`.

spe_program_handle_t* loaded_program

Definition at line 99 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_program_load()`, and `_base_spe_program_load_complete()`.

void* mem_mmap_base

Definition at line 85 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_handle_library_callback()`, `_base_spe_ls_area_get()`, and `_base_spe_program_load()`.

void* mfc_mmap_base

Definition at line 86 of file spebase.h.

Referenced by `_base_spe_context_create()`, and `_base_spe_ps_area_get()`.

void* mssync_mmap_base

Definition at line 87 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_mssync_start()`, `_base_spe_mssync_status()`, and `_base_spe_ps_area_get()`.

void* psmap_mmap_base

Definition at line 84 of file spebase.h.

Referenced by `_base_spe_context_create()`.

void* signal1_mmap_base

Definition at line 89 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_ps_area_get()`, and `_base_spe_signal_write()`.

void* signal2_mmap_base

Definition at line 90 of file spebase.h.

Referenced by `_base_spe_context_create()`, `_base_spe_ps_area_get()`, and `_base_spe_signal_write()`.

int spe_fds_array[NUM_MBOX_FDS]

Definition at line 77 of file spebase.h.

Referenced by `_base_spe_close_if_open()`, `_base_spe_context_create()`, and `_base_spe_open_if_closed()`.

int spe_fds_refcount[[NUM_MBOX_FDS](#)]

Definition at line 78 of file [spebase.h](#).

Referenced by [_base_spe_close_if_open\(\)](#), and [_base_spe_open_if_closed\(\)](#).

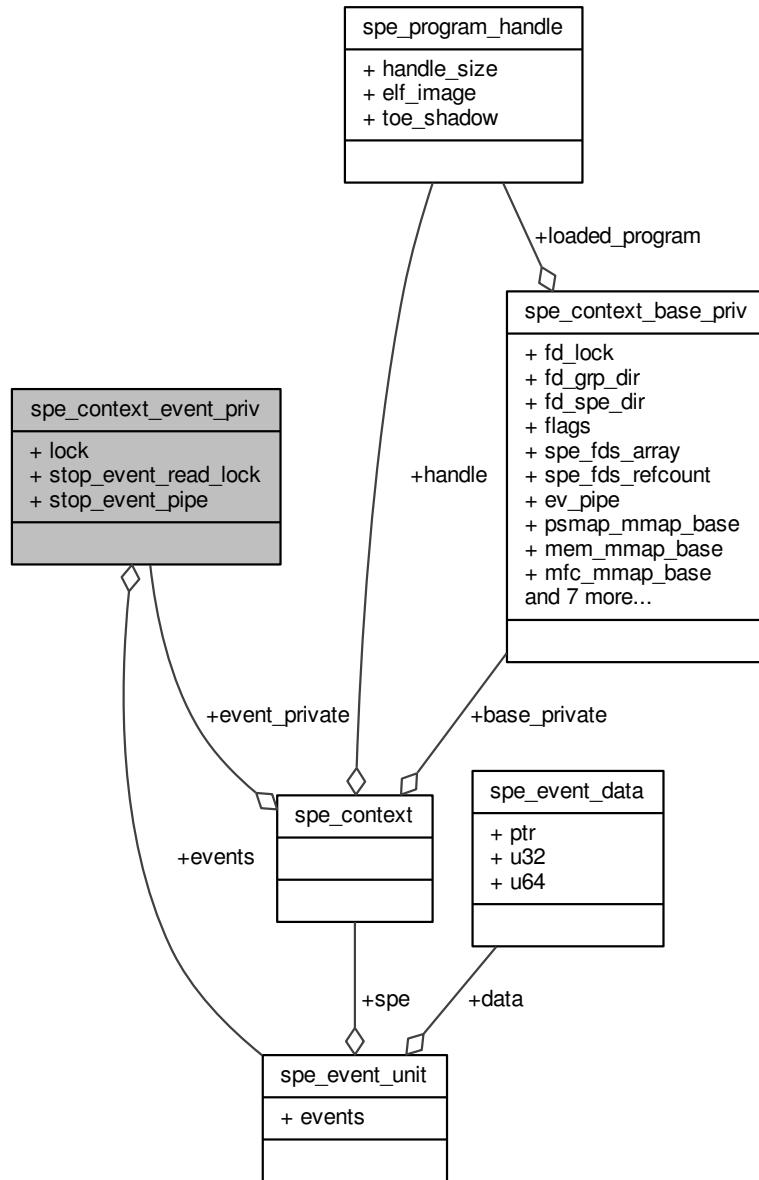
The documentation for this struct was generated from the following file:

- [spebase.h](#)

2.7 [spe_context_event_priv](#) Struct Reference

```
#include <speevent.h>
```

Collaboration diagram for `spe_context_event_priv`:



Data Fields

- `pthread_mutex_t` [lock](#)
- `pthread_mutex_t` [stop_event_read_lock](#)
- `int` [stop_event_pipe](#) [2]
- `spe_event_unit_t` [events](#) [`__NUM_SPE_EVENT_TYPES`]

2.7.1 Detailed Description

Definition at line 35 of file speevent.h.

2.7.2 Field Documentation

spe_event_unit_t events[[__NUM_SPE_EVENT_TYPES](#)]

Definition at line 40 of file speevent.h.

Referenced by [_event_spe_context_initialize\(\)](#), [_event_spe_event_handler_deregister\(\)](#), and [_event_spe_event_handler_register\(\)](#).

pthread_mutex_t lock

Definition at line 37 of file speevent.h.

Referenced by [_event_spe_context_finalize\(\)](#), and [_event_spe_context_initialize\(\)](#).

int stop_event_pipe[2]

Definition at line 39 of file speevent.h.

Referenced by [_event_spe_context_finalize\(\)](#), [_event_spe_context_initialize\(\)](#), [_event_spe_context_run\(\)](#), [_event_spe_event_handler_deregister\(\)](#), [_event_spe_event_handler_register\(\)](#), and [_event_spe_stop_info_read\(\)](#).

pthread_mutex_t stop_event_read_lock

Definition at line 38 of file speevent.h.

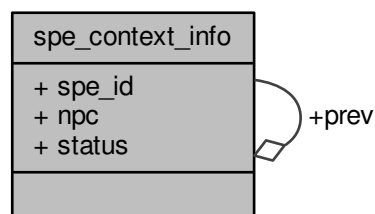
Referenced by [_event_spe_context_finalize\(\)](#), [_event_spe_context_initialize\(\)](#), and [_event_spe_stop_info_read\(\)](#).

The documentation for this struct was generated from the following file:

- [speevent.h](#)

2.8 spe_context_info Struct Reference

Collaboration diagram for [spe_context_info](#):



Data Fields

- int [spe_id](#)
- unsigned int [npc](#)
- unsigned int [status](#)
- struct [spe_context_info](#) * [prev](#)

2.8.1 Detailed Description

Definition at line 40 of file run.c.

2.8.2 Field Documentation

unsigned int npc

Definition at line 42 of file run.c.

Referenced by `_base_spe_context_run()`.

struct [spe_context_info](#)* prev

Definition at line 44 of file run.c.

Referenced by `_base_spe_context_run()`.

int spe_id

Definition at line 41 of file run.c.

Referenced by `_base_spe_context_run()`.

unsigned int status

Definition at line 43 of file run.c.

Referenced by `_base_spe_context_run()`.

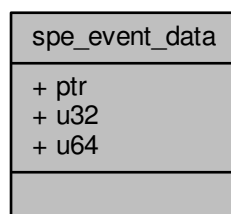
The documentation for this struct was generated from the following file:

- [run.c](#)

2.9 spe_event_data Union Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for `spe_event_data`:



Data Fields

- void * [ptr](#)
- unsigned int [u32](#)
- unsigned long long [u64](#)

2.9.1 Detailed Description

`spe_event_data_t` User data to be associated with an event

Definition at line 143 of file `libspe2-types.h`.

2.9.2 Field Documentation

void* ptr

Definition at line 145 of file `libspe2-types.h`.

Referenced by `_event_spe_event_handler_register()`.

unsigned int u32

Definition at line 146 of file `libspe2-types.h`.

unsigned long long u64

Definition at line 147 of file `libspe2-types.h`.

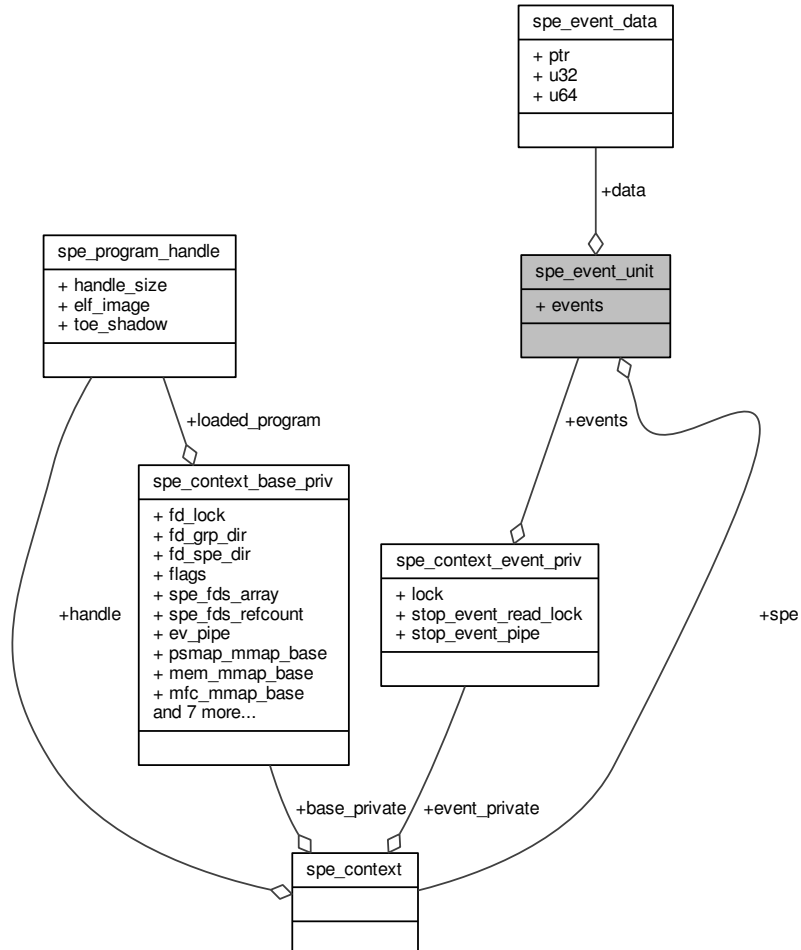
The documentation for this union was generated from the following file:

- [libspe2-types.h](#)

2.10 spe_event_unit Struct Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for `spe_event_unit`:



Data Fields

- unsigned int `events`
- `spe_context_ptr_t` `spe`
- `spe_event_data_t` `data`

2.10.1 Detailed Description

`spe_event_t`

Definition at line 152 of file `libspe2-types.h`.

2.10.2 Field Documentation

`spe_event_data_t` `data`

Definition at line 156 of file `libspe2-types.h`.

Referenced by `_event_spe_event_handler_register()`.

unsigned int events

Definition at line 154 of file `libspe2-types.h`.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

spe_context_ptr_t spe

Definition at line 155 of file `libspe2-types.h`.

Referenced by `_event_spe_context_initialize()`, `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_register()`, and `_event_spe_event_wait()`.

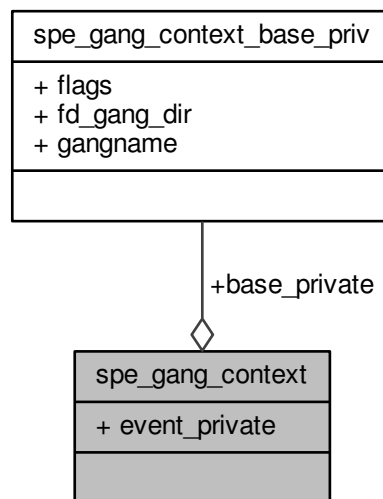
The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

2.11 spe_gang_context Struct Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for `spe_gang_context`:



Data Fields

- struct [spe_gang_context_base_priv](#) * `base_private`
- struct `spe_gang_context_event_priv` * [event_private](#)

2.11.1 Detailed Description

SPE gang context The SPE gang context is one of the base data structures for the libspe2 implementation. It holds all persistent information about a group of SPE contexts that should be treated as a gang, i.e., be execute together with certain properties. This data structure should not be accessed directly, but the application uses a pointer to an SPE gang context as an identifier for the SPE gang it is dealing with through libspe2 API calls.

Definition at line 94 of file libspe2-types.h.

2.11.2 Field Documentation

struct spe_gang_context_base_priv* base_private

Definition at line 99 of file libspe2-types.h.

Referenced by `_base_spe_context_create()`, and `_base_spe_gang_context_create()`.

struct spe_gang_context_event_priv* event_private

Definition at line 100 of file libspe2-types.h.

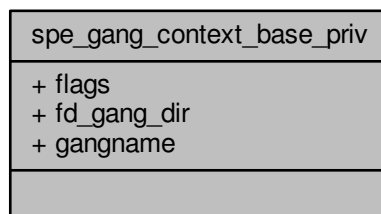
The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

2.12 spe_gang_context_base_priv Struct Reference

```
#include <spebase.h>
```

Collaboration diagram for `spe_gang_context_base_priv`:



Data Fields

- unsigned int [flags](#)
- int [fd_gang_dir](#)
- char [gangname](#) [256]

2.12.1 Detailed Description

spe_context: This holds the persistant information of a SPU instance it is created by `spe_create_context()`

Definition at line 150 of file spebase.h.

2.12.2 Field Documentation

int fd_gang_dir

Definition at line 156 of file spebase.h.

unsigned int flags

Definition at line 153 of file spebase.h.

char gangname[256]

Definition at line 158 of file spebase.h.

Referenced by `_base_spe_context.create()`, and `_base_spe_gang_context.create()`.

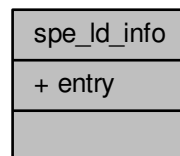
The documentation for this struct was generated from the following file:

- [spebase.h](#)

2.13 spe_ld_info Struct Reference

```
#include <elf_loader.h>
```

Collaboration diagram for `spe_ld_info`:



Data Fields

- unsigned int [entry](#)

2.13.1 Detailed Description

Definition at line 34 of file `elf_loader.h`.

2.13.2 Field Documentation

unsigned int entry

Definition at line 36 of file `elf_loader.h`.

Referenced by `_base_spe_load_spe_elf()`, and `_base_spe_program_load()`.

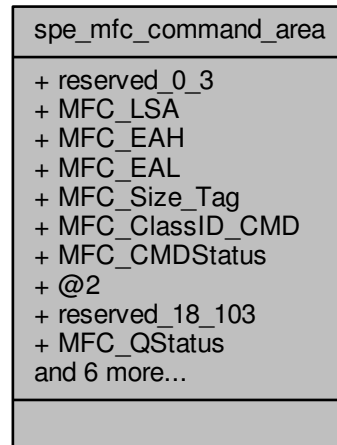
The documentation for this struct was generated from the following file:

- [elf_loader.h](#)

2.14 spe_mfc_command_area Struct Reference

```
#include <cbea_map.h>
```

Collaboration diagram for spe_mfc_command_area:



Data Fields

- unsigned char [reserved_0_3](#) [4]
- unsigned int [MFC_LSA](#)
- unsigned int [MFC_EAH](#)
- unsigned int [MFC_EAL](#)
- unsigned int [MFC_Size_Tag](#)
- union {
 - unsigned int [MFC_ClassID_CMD](#)
 - unsigned int [MFC_CMDStatus](#)
 };
- unsigned char [reserved_18_103](#) [236]
- unsigned int [MFC_QStatus](#)
- unsigned char [reserved_108_203](#) [252]
- unsigned int [Prxy_QueryType](#)
- unsigned char [reserved_208_21B](#) [20]
- unsigned int [Prxy_QueryMask](#)
- unsigned char [reserved_220_22B](#) [12]
- unsigned int [Prxy_TagStatus](#)

2.14.1 Detailed Description

Definition at line 34 of file `cbea_map.h`.

2.14.2 Field Documentation

union { ... }

unsigned int MFC_ClassID_CMD

Definition at line 41 of file cbea_map.h.

unsigned int MFC_CMDStatus

Definition at line 42 of file cbea_map.h.

unsigned int MFC_EAH

Definition at line 37 of file cbea_map.h.

unsigned int MFC_EAL

Definition at line 38 of file cbea_map.h.

unsigned int MFC_LSA

Definition at line 36 of file cbea_map.h.

unsigned int MFC_QStatus

Definition at line 45 of file cbea_map.h.

unsigned int MFC_Size.Tag

Definition at line 39 of file cbea_map.h.

unsigned int Prxy_QueryMask

Definition at line 49 of file cbea_map.h.

unsigned int Prxy_QueryType

Definition at line 47 of file cbea_map.h.

unsigned int Prxy_TagStatus

Definition at line 51 of file cbea_map.h.

unsigned char reserved_0_3[4]

Definition at line 35 of file cbea_map.h.

unsigned char reserved_108_203[252]

Definition at line 46 of file cbea_map.h.

unsigned char reserved_18_103[236]

Definition at line 44 of file cbea_map.h.

unsigned char reserved_208_21B[20]

Definition at line 48 of file cbea_map.h.

unsigned char reserved_220_22B[12]

Definition at line 50 of file cbea_map.h.

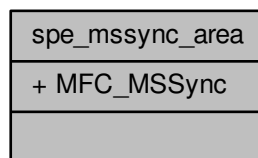
The documentation for this struct was generated from the following file:

- [cbea_map.h](#)

2.15 spe_mssync_area Struct Reference

```
#include <cbea_map.h>
```

Collaboration diagram for spe_mssync_area:



Data Fields

- unsigned int [MFC_MSSync](#)

2.15.1 Detailed Description

Definition at line 30 of file cbea_map.h.

2.15.2 Field Documentation

unsigned int MFC_MSSync

Definition at line 31 of file cbea_map.h.

Referenced by `_base_spe_mssync_start()`, and `_base_spe_mssync_status()`.

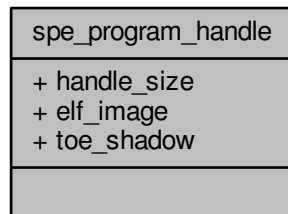
The documentation for this struct was generated from the following file:

- [cbea_map.h](#)

2.16 spe_program_handle Struct Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for `spe_program_handle`:



Data Fields

- unsigned int [handle_size](#)
- void * [elf_image](#)
- void * [toe_shadow](#)

2.16.1 Detailed Description

SPE program handle Structure [spe_program_handle](#) per CESOF specification libspe2 applications usually only keep a pointer to the program handle and do not use the structure directly.

Definition at line 43 of file `libspe2-types.h`.

2.16.2 Field Documentation

void* `elf_image`

Definition at line 50 of file `libspe2-types.h`.

Referenced by `_base_spe_image_close()`, `_base_spe_image_open()`, `_base_spe_load_spe_elf()`, `_base_spe_parse_isolated_elf()`, `_base_spe_program_load_complete()`, `_base_spe_toe_ear()`, and `_base_spe_verify_spe_elf_image()`.

unsigned int `handle_size`

Definition at line 49 of file `libspe2-types.h`.

Referenced by `_base_spe_image_open()`.

void* `toe_shadow`

Definition at line 51 of file `libspe2-types.h`.

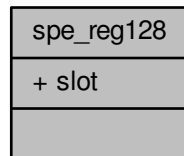
Referenced by `_base_spe_image_close()`, `_base_spe_image_open()`, and `_base_spe_toe_ear()`.

The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

2.17 spe_reg128 Struct Reference

```
#include <handler_utils.h>
Collaboration diagram for spe_reg128:
```



Data Fields

- unsigned int [slot](#) [4]

2.17.1 Detailed Description

Definition at line 23 of file handler_utils.h.

2.17.2 Field Documentation

unsigned int slot[4]

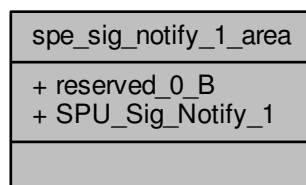
Definition at line 24 of file handler_utils.h.

The documentation for this struct was generated from the following file:

- [handler_utils.h](#)

2.18 spe_sig_notify_1_area Struct Reference

```
#include <cbea_map.h>
Collaboration diagram for spe_sig_notify_1_area:
```



Data Fields

- unsigned char [reserved_0_B](#) [12]
- unsigned int [SPU_Sig_Notify_1](#)

2.18.1 Detailed Description

Definition at line 69 of file cbea_map.h.

2.18.2 Field Documentation

unsigned char reserved_0_B[12]

Definition at line 70 of file cbea_map.h.

unsigned int SPU_Sig_Notify_1

Definition at line 71 of file cbea_map.h.

Referenced by `_base_spe_signal.write()`.

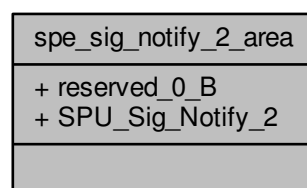
The documentation for this struct was generated from the following file:

- [cbea_map.h](#)

2.19 spe_sig_notify_2_area Struct Reference

```
#include <cbea_map.h>
```

Collaboration diagram for `spe_sig_notify_2_area`:



Data Fields

- unsigned char [reserved_0_B](#) [12]
- unsigned int [SPU_Sig_Notify_2](#)

2.19.1 Detailed Description

Definition at line 74 of file cbea_map.h.

2.19.2 Field Documentation

unsigned char reserved_0_B[12]

Definition at line 75 of file cbea_map.h.

unsigned int SPU_Sig_Notify_2

Definition at line 76 of file cbea_map.h.

Referenced by `_base_spe_signal_write()`.

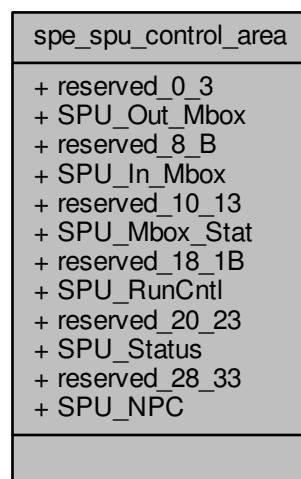
The documentation for this struct was generated from the following file:

- [cbea_map.h](#)

2.20 spe_spu_control_area Struct Reference

```
#include <cbea_map.h>
```

Collaboration diagram for `spe_spu_control_area`:



Data Fields

- unsigned char [reserved_0_3](#) [4]
- unsigned int [SPU_Out_Mbox](#)
- unsigned char [reserved_8_B](#) [4]
- unsigned int [SPU_In_Mbox](#)
- unsigned char [reserved_10_13](#) [4]
- unsigned int [SPU_Mbox_Stat](#)
- unsigned char [reserved_18_1B](#) [4]
- unsigned int [SPU_RunCntl](#)
- unsigned char [reserved_20_23](#) [4]

- unsigned int [SPU_Status](#)
- unsigned char [reserved_28_33](#) [12]
- unsigned int [SPU_NPC](#)

2.20.1 Detailed Description

Definition at line 54 of file cbea_map.h.

2.20.2 Field Documentation

unsigned char reserved_0_3[4]

Definition at line 55 of file cbea_map.h.

unsigned char reserved_10_13[4]

Definition at line 59 of file cbea_map.h.

unsigned char reserved_18_1B[4]

Definition at line 61 of file cbea_map.h.

unsigned char reserved_20_23[4]

Definition at line 63 of file cbea_map.h.

unsigned char reserved_28_33[12]

Definition at line 65 of file cbea_map.h.

unsigned char reserved_8_B[4]

Definition at line 57 of file cbea_map.h.

unsigned int SPU_In_Mbox

Definition at line 58 of file cbea_map.h.

unsigned int SPU_Mbox_Stat

Definition at line 60 of file cbea_map.h.

Referenced by `_base_spe_in_mbox_status()`, `_base_spe_out_intr_mbox_status()`, and `_base_spe_out_mbox_status()`.

unsigned int SPU_NPC

Definition at line 66 of file cbea_map.h.

unsigned int SPU_Out_Mbox

Definition at line 56 of file cbea_map.h.

unsigned int SPU_RunCntl

Definition at line 62 of file cbea_map.h.

unsigned int SPU_Status

Definition at line 64 of file cbea_map.h.

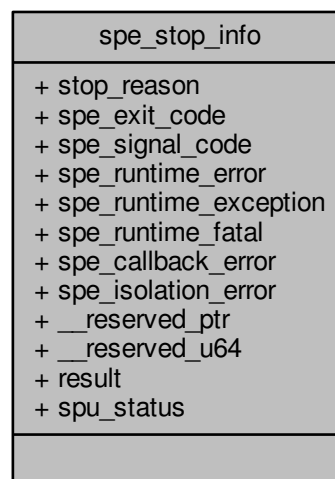
The documentation for this struct was generated from the following file:

- [cbea_map.h](#)

2.21 spe_stop_info Struct Reference

```
#include <libspe2-types.h>
```

Collaboration diagram for spe_stop_info:

**Data Fields**

- unsigned int [stop_reason](#)
- union {
 - int [spe_exit_code](#)
 - int [spe_signal_code](#)
 - int [spe_runtime_error](#)
 - int [spe_runtime_exception](#)
 - int [spe_runtime_fatal](#)
 - int [spe_callback_error](#)
 - int [spe_isolation_error](#)
 - void * [__reserved_ptr](#)
 - unsigned long long [__reserved_u64](#)
- int [spu_status](#)

2.21.1 Detailed Description

`spe_stop_info_t`

Definition at line 118 of file `libspe2-types.h`.

2.21.2 Field Documentation

`void* __reserved_ptr`

Definition at line 129 of file `libspe2-types.h`.

`unsigned long long __reserved_u64`

Definition at line 130 of file `libspe2-types.h`.

`union { ... } result`

Referenced by `_base_spe_context_run()`.

`int spe_callback_error`

Definition at line 126 of file `libspe2-types.h`.

Referenced by `_base_spe_context_run()`.

`int spe_exit_code`

Definition at line 121 of file `libspe2-types.h`.

Referenced by `_base_spe_context_run()`.

`int spe_isolation_error`

Definition at line 127 of file `libspe2-types.h`.

Referenced by `_base_spe_context_run()`.

`int spe_runtime_error`

Definition at line 123 of file `libspe2-types.h`.

Referenced by `_base_spe_context_run()`.

`int spe_runtime_exception`

Definition at line 124 of file `libspe2-types.h`.

Referenced by `_base_spe_context_run()`.

`int spe_runtime_fatal`

Definition at line 125 of file `libspe2-types.h`.

Referenced by `_base_spe_context_run()`.

`int spe_signal_code`

Definition at line 122 of file `libspe2-types.h`.

Referenced by `_base_spe_context_run()`.

int spu_status

Definition at line 132 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

unsigned int stop_reason

Definition at line 119 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

The documentation for this struct was generated from the following file:

- [libspe2-types.h](#)

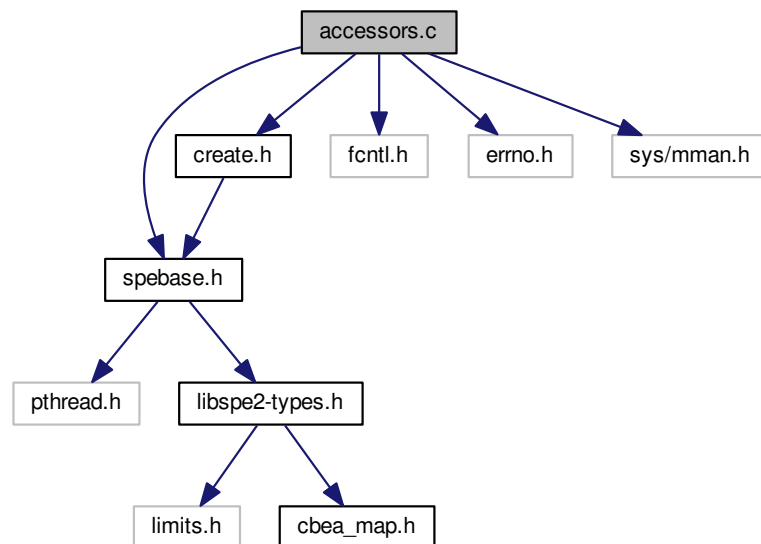
Chapter 3

File Documentation

3.1 `accessors.c` File Reference

```
#include "spebase.h"  
#include "create.h"  
#include <fcntl.h>  
#include <errno.h>  
#include <sys/mman.h>
```

Include dependency graph for `accessors.c`:



Functions

- `void * _base_spe_ps_area_get (spe_context_ptr_t spe, enum ps_area area)`
- `void * _base_spe_ls_area_get (spe_context_ptr_t spe)`
- `__attribute__((noinline))`

- `int __base_spe_event_source_acquire (spe_context_ptr_t spe, enum fd_name fdesc)`
- `void __base_spe_event_source_release (struct spe_context *spe, enum fd_name fdesc)`
- `int __base_spe_spe_dir_get (spe_context_ptr_t spe)`
- `int __base_spe_stop_event_source_get (spe_context_ptr_t spe)`
- `int __base_spe_stop_event_target_get (spe_context_ptr_t spe)`
- `int __base_spe_ls_size_get (spe_context_ptr_t spe)`

3.1.1 Function Documentation

`__attribute__((noinline))`

Definition at line 69 of file `accessors.c`.

```
70 {
71     return;
72 }
```

`int __base_spe_event_source_acquire (spe_context_ptr_t spe, enum fd_name fdesc)`

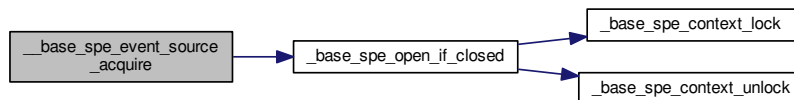
Definition at line 74 of file `accessors.c`.

References `_base_spe_open_if_closed()`.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

```
75 {
76     return _base_spe_open_if_closed(spe, fdesc, 0);
77 }
```

Here is the call graph for this function:



`void __base_spe_event_source_release (struct spe_context * spectx, enum fd_name fdesc)`

`__base_spe_event_source_release` releases the file descriptor to the specified event source

Parameters

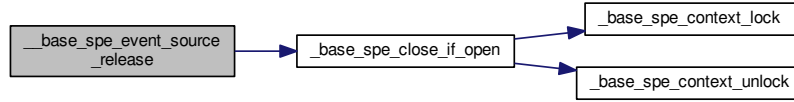
<i>spectx</i>	Specifies the SPE context
<i>fdesc</i>	Specifies the event source

Definition at line 79 of file `accessors.c`.

References `_base_spe_close_if_open()`.

```
80 {
81     _base_spe_close_if_open(spe, fdesc);
82 }
```


Here is the call graph for this function:



int __base_spe_spe_dir_get (spe_context_ptr_t spe)

Definition at line 84 of file accessors.c.

References spe_context::base_private, and spe_context.base_priv::fd_spe_dir.

```

85 {
86     return spe->base_private->fd_spe_dir;
87 }
  
```

int __base_spe_stop_event_source_get (spe_context_ptr_t spe)

speevent users read from this end

Definition at line 92 of file accessors.c.

References spe_context::base_private, and spe_context.base_priv::ev_pipe.

```

93 {
94     return spe->base_private->ev_pipe[1];
95 }
  
```

int __base_spe_stop_event_target_get (spe_context_ptr_t spe)

speevent writes to this end

Definition at line 100 of file accessors.c.

References spe_context::base_private, and spe_context.base_priv::ev_pipe.

```

101 {
102     return spe->base_private->ev_pipe[0];
103 }
  
```

void* _base_spe_ls_area_get (spe_context_ptr_t spe)

Definition at line 64 of file accessors.c.

References spe_context::base_private, and spe_context.base_priv::mem_mmap_base.

```

65 {
66     return spe->base_private->mem_mmap_base;
67 }
  
```

int _base_spe_ls_size_get (spe_context_ptr_t spe)

_base_spe_ls_size_get returns the size of the local store area

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

Definition at line 105 of file accessors.c.

References LS_SIZE.

```

106 {
107     return LS_SIZE;
108 }
```

void* _base_spe_ps_area_get (spe_context_ptr_t spe, enum ps_area area)

Definition at line 30 of file accessors.c.

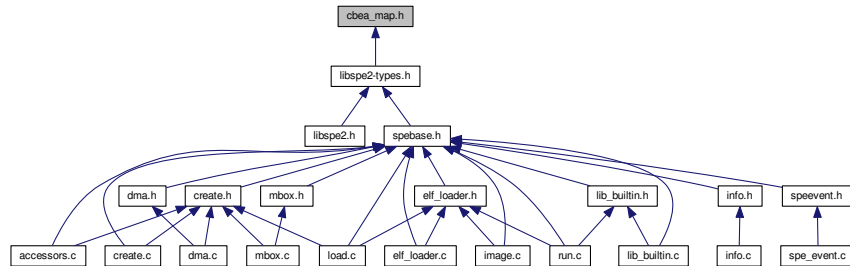
References spe_context::base_private, spe_context_base_priv::cntl_mmap_base, spe_context_base_priv::mfc_mmap_base, spe_context_base_priv::mssync_mmap_base, spe_context_base_priv::signal1_mmap_base, spe_context_base_priv::signal2_mmap_base, SPE_CONTROL_AREA, SPE_MFC_COMMAND_AREA, SPE_MSSYNC_AREA, SPE_SIG_NOTIFY_1_AREA, and SPE_SIG_NOTIFY_2_AREA.

```

31 {
32     void *ptr;
33
34     switch (area) {
35         case SPE_MSSYNC_AREA:
36             ptr = spe->base_private->mssync_mmap_base;
37             break;
38         case SPE_MFC_COMMAND_AREA:
39             ptr = spe->base_private->mfc_mmap_base;
40             break;
41         case SPE_CONTROL_AREA:
42             ptr = spe->base_private->cntl_mmap_base;
43             break;
44         case SPE_SIG_NOTIFY_1_AREA:
45             ptr = spe->base_private->signal1_mmap_base;
46             break;
47         case SPE_SIG_NOTIFY_2_AREA:
48             ptr = spe->base_private->signal2_mmap_base;
49             break;
50         default:
51             errno = EINVAL;
52             return NULL;
53             break;
54     }
55
56     if (ptr == MAP_FAILED) {
57         errno = EACCES;
58         return NULL;
59     }
60
61     return ptr;
62 }
```

3.2 cbea_map.h File Reference

This graph shows which files directly or indirectly include this file:



Data Structures

- struct [spe_mssync_area](#)
- struct [spe_mfc_command_area](#)
- struct [spe_spu_control_area](#)
- struct [spe_sig_notify_1_area](#)
- struct [spe_sig_notify_2_area](#)

Typedefs

- typedef struct [spe_mssync_area](#) [spe_mssync_area_t](#)
- typedef struct [spe_mfc_command_area](#) [spe_mfc_command_area_t](#)
- typedef struct [spe_spu_control_area](#) [spe_spu_control_area_t](#)
- typedef struct [spe_sig_notify_1_area](#) [spe_sig_notify_1_area_t](#)
- typedef struct [spe_sig_notify_2_area](#) [spe_sig_notify_2_area_t](#)

3.2.1 Typedef Documentation

```
typedef struct spe_mfc_command_area spe_mfc_command_area_t
```

```
typedef struct spe_mssync_area spe_mssync_area_t
```

```
typedef struct spe_sig_notify_1_area spe_sig_notify_1_area_t
```

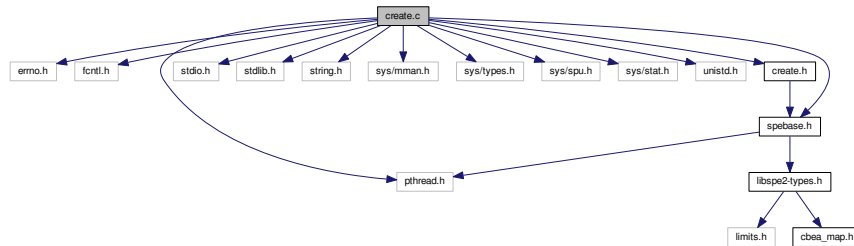
```
typedef struct spe_sig_notify_2_area spe_sig_notify_2_area_t
```

```
typedef struct spe_spu_control_area spe_spu_control_area_t
```

3.3 create.c File Reference

```
#include <errno.h>
#include <fcntl.h>
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/spu.h>
#include <sys/stat.h>
#include <unistd.h>
#include "create.h"
#include "spebase.h"
```

Include dependency graph for create.c:



Data Structures

- struct [fd_attr](#)

Functions

- void [_base_spe_context_lock](#) ([spe_context_ptr_t](#) spe, enum [fd_name](#) fdesc)
- void [_base_spe_context_unlock](#) ([spe_context_ptr_t](#) spe, enum [fd_name](#) fdesc)
- int [_base_spe_open_if_closed](#) (struct [spe_context](#) *spe, enum [fd_name](#) fdesc, int locked)
- void [_base_spe_close_if_open](#) (struct [spe_context](#) *spe, enum [fd_name](#) fdesc)
- [spe_context_ptr_t](#) [_base_spe_context_create](#) (unsigned int flags, [spe_gang_context_ptr_t](#) gctx, [spe_context_ptr_t](#) aff_spe)
- [spe_gang_context_ptr_t](#) [_base_spe_gang_context_create](#) (unsigned int flags)
- int [_base_spe_context_destroy](#) ([spe_context_ptr_t](#) spe)
- int [_base_spe_gang_context_destroy](#) ([spe_gang_context_ptr_t](#) gctx)

3.3.1 Function Documentation

void _base_spe_close_if_open (struct spe_context * spe, enum fd_name fdesc)

Definition at line 125 of file create.c.

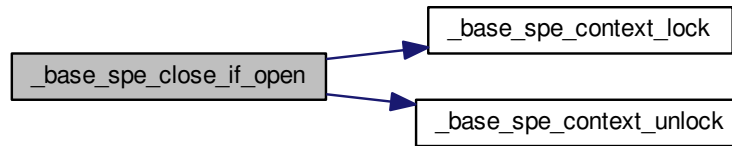
References `_base_spe_context_lock()`, `_base_spe_context_unlock()`, `spe_context::base_private`, `spe_context::base_priv::spe_fds_array`, and `spe_context::base_priv::spe_fds_refcount`.

Referenced by `__base_spe_event_source_release()`, and `_base_spe_signal_write()`.

```

126 {
127     _base_spe_context_lock(spe, fdesc);
128
129     if (spe->base_private->spe_fds_array[(int)fdesc] != -1 &&
130         spe->base_private->spe_fds_refcount[(int)fdesc] == 1) {
131
132         spe->base_private->spe_fds_refcount[(int)fdesc]--;
133         close(spe->base_private->spe_fds_array[(int)fdesc]);
134
135         spe->base_private->spe_fds_array[(int)fdesc] = -1;
136     } else if (spe->base_private->spe_fds_refcount[(int)fdesc] > 0) {
137         spe->base_private->spe_fds_refcount[(int)fdesc]--;
138     }
139
140     _base_spe_context_unlock(spe, fdesc);
141 }
```

Here is the call graph for this function:



spe_context_ptr_t _base_spe_context_create (unsigned int flags, spe_gang_context_ptr_t gctx, spe_context_ptr_t aff_spe)

`_base_spe_context_create` creates a single SPE context, i.e., the corresponding directory is created in SPUPS either as a subdirectory of a gang or individually (maybe this is best considered a gang of one)

Parameters

<i>flags</i>	
<i>gctx</i>	specify NULL if not belonging to a gang
<i>aff_spe</i>	specify NULL to skip affinity information

Definition at line 183 of file create.c.

References `_base_spe_emulated_loader_present()`, `spe_context::base_private`, `spe_gang_context::base_private`, `spe_context::base_priv::cntl_mmap_base`, `CNTL_OFFSET`, `CNTL_SIZE`, `DEBUG_PRINTF`, `spe_context::base_priv::fd_lock`, `spe_context::base_priv::fd_spe_dir`, `spe_context::base_priv::flags`, `spe_gang_context::base_priv::gangname`, `spe_context::base_priv::loaded_program`, `LS_SIZE`, `spe_context::base_priv::mem_mmap_base`, `spe_context::base_priv::mfc_mmap_base`, `MFC_OFFSET`, `MFC_SIZE`, `MSS_SIZE`, `spe_context::base_priv::mssync_mmap_base`, `MSSYNC_OFFSET`, `NUM_MBOX_FDS`, `spe_context::base_priv::psmap_mmap_base`, `PSMAP_SIZE`, `spe_context::base_priv::signal1_mmap_base`, `SIGNAL1_OFFSET`, `spe_context::base_priv::signal2_mmap_base`, `SIGNAL2_OFFSET`, `SIGNAL_SIZE`, `SPE_AFFINITY_MEMORY`, `SPE_CFG_`

SIGNOTIFY1_OR, SPE_CFG_SIGNOTIFY2_OR, SPE_EVENTS_ENABLE, spe_context_base_priv::spe_fds_array, SPE_ISOLATE, SPE_ISOLATE_EMULATE, and SPE_MAP_PS.

```

185 {
186     char pathname[256];
187     int i, aff_spe_fd = 0;
188     unsigned int spu_createflags = 0;
189     struct spe_context *spe = NULL;
190     struct spe_context_base_priv *priv;
191
192     /* We need a loader present to run in emulated isolated mode */
193     if (flags & SPE_ISOLATE_EMULATE
194         && !_base_spe_emulated_loader_present()) {
195         errno = EINVAL;
196         return NULL;
197     }
198
199     /* Put some sane defaults into the SPE context */
200     spe = malloc(sizeof(*spe));
201     if (!spe) {
202         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
203         return NULL;
204     }
205     memset(spe, 0, sizeof(*spe));
206
207     spe->base_private = malloc(sizeof(*spe->base_private));
208     if (!spe->base_private) {
209         DEBUG_PRINTF("ERROR: Could not allocate "
210                     "spe->base-private context.\n");
211         free(spe);
212         return NULL;
213     }
214
215     /* just a convenience variable */
216     priv = spe->base_private;
217
218     priv->fd_spe_dir = -1;
219     priv->mem_mmap_base = MAP_FAILED;
220     priv->psmap_mmap_base = MAP_FAILED;
221     priv->mssync_mmap_base = MAP_FAILED;
222     priv->mfc_mmap_base = MAP_FAILED;
223     priv->cntl_mmap_base = MAP_FAILED;
224     priv->signal1_mmap_base = MAP_FAILED;
225     priv->signal2_mmap_base = MAP_FAILED;
226     priv->loaded_program = NULL;
227
228     for (i = 0; i < NUM_MBOX_FDS; i++) {
229         priv->spe_fds_array[i] = -1;
230         pthread_mutex_init(&priv->fd_lock[i], NULL);
231     }
232
233     /* initialise spu_createflags */
234     if (flags & SPE_ISOLATE) {
235         flags |= SPE_MAP_PS;
236         spu_createflags |= SPU_CREATE_ISOLATE | SPU_CREATE_NOSCHED;
237     }
238
239     if (flags & SPE_EVENTS_ENABLE)
240         spu_createflags |= SPU_CREATE_EVENTS_ENABLED;
241
242     if (aff_spe)
243         spu_createflags |= SPU_CREATE_AFFINITY_SPU;
244
245     if (flags & SPE_AFFINITY_MEMORY)
246         spu_createflags |= SPU_CREATE_AFFINITY_MEM;
247
248     /* Make the SPUFS directory for the SPE */
249     if (gctx == NULL)
250         sprintf(pathname, "/spu/spthread-%i-%lu",
251                 getpid(), (unsigned long)spe);
252     else
253         sprintf(pathname, "/spu/%s/spthread-%i-%lu",
254                 gctx->base_private->gangname, getpid(),
255                 (unsigned long)spe);
256
257     if (aff_spe)
258         aff_spe_fd = aff_spe->base_private->fd_spe_dir;
259
260     priv->fd_spe_dir = spu_create(pathname, spu_createflags,
261                                S_IRUSR | S_IWUSR | S_IXUSR, aff_spe_fd);

```

```

262
263     if (priv->fd_spe_dir < 0) {
264         int errno_saved = errno; /* save errno to prevent being overwritten */
265         DEBUG_PRINTF("ERROR: Could not create SPE %s\n", pathname);
266         perror("spu.create()");
267         free_spe_context(spe);
268         /* we mask most errors, but leave ENODEV, etc */
269         switch (errno_saved) {
270             case ENOTSUP:
271             case EEXIST:
272             case EINVAL:
273             case EBUSY:
274             case EPERM:
275             case ENODEV:
276                 errno = errno_saved; /* restore errno */
277                 break;
278             default:
279                 errno = EFAULT;
280                 break;
281         }
282         return NULL;
283     }
284
285     priv->flags = flags;
286
287     /* Map the required areas into process memory */
288     priv->mem_mmap_base = mapfileat(priv->fd_spe_dir, "mem",
LS.SIZE);
289     if (priv->mem_mmap_base == MAP_FAILED) {
290         DEBUG_PRINTF("ERROR: Could not map SPE memory.\n");
291         free_spe_context(spe);
292         errno = ENOMEM;
293         return NULL;
294     }
295
296     if (flags & SPE_MAP_PS) {
297         /* It's possible to map the entire problem state area with
298          * one mmap - try this first */
299         priv->psmap_mmap_base = mapfileat(priv->
fd_spe_dir,
300                                         "psmap", PSMAP_SIZE);
301
302         if (priv->psmap_mmap_base != MAP_FAILED) {
303             priv->mssync_mmap_base =
304                 priv->psmap_mmap_base +
MSSYNC_OFFSET;
305             priv->mfc_mmap_base =
306                 priv->psmap_mmap_base +
MFC_OFFSET;
307             priv->cntl_mmap_base =
308                 priv->psmap_mmap_base +
CNTL_OFFSET;
309             priv->signal1_mmap_base =
310                 priv->psmap_mmap_base +
SIGNAL1_OFFSET;
311             priv->signal2_mmap_base =
312                 priv->psmap_mmap_base +
SIGNAL2_OFFSET;
313         } else {
314             /* map each region separately */
315             priv->mfc_mmap_base =
316                 mapfileat(priv->fd_spe_dir, "mfc",
MFC_SIZE);
317             priv->mssync_mmap_base =
318                 mapfileat(priv->fd_spe_dir, "mss",
MSS_SIZE);
319             priv->cntl_mmap_base =
320                 mapfileat(priv->fd_spe_dir, "cntl",
CNTL_SIZE);
321             priv->signal1_mmap_base =
322                 mapfileat(priv->fd_spe_dir, "signal1",
SIGNAL_SIZE);
323             priv->signal2_mmap_base =
324                 mapfileat(priv->fd_spe_dir, "signal2",
SIGNAL_SIZE);
325
326             if (priv->mfc_mmap_base == MAP_FAILED ||
327                 priv->cntl_mmap_base == MAP_FAILED ||
328                 priv->signal1_mmap_base == MAP_FAILED ||
329                 priv->signal2_mmap_base == MAP_FAILED ||
330                 priv->mssync_mmap_base == MAP_FAILED ||
331                 priv->psmap_mmap_base == MAP_FAILED) {

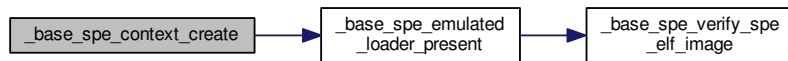
```

```

332         priv->signal2_mmap_base == MAP_FAILED) {
333             DEBUG_PRINTF("ERROR: Could not map SPE "
334                 "PS memory.\n");
335             free_spe_context(spe);
336             errno = ENOMEM;
337             return NULL;
338         }
339     }
340 }
341
342 if (flags & SPE_CFG.SIGNOTIFY1_OR) {
343     if (setsignotify(priv->fd.spe_dir, "signal1.type")) {
344         DEBUG_PRINTF("ERROR: Could not open SPE "
345             "signal1.type file.\n");
346         free_spe_context(spe);
347         errno = EFAULT;
348         return NULL;
349     }
350 }
351
352 if (flags & SPE_CFG.SIGNOTIFY2_OR) {
353     if (setsignotify(priv->fd.spe_dir, "signal2.type")) {
354         DEBUG_PRINTF("ERROR: Could not open SPE "
355             "signal2.type file.\n");
356         free_spe_context(spe);
357         errno = EFAULT;
358         return NULL;
359     }
360 }
361
362 return spe;
363 }

```

Here is the call graph for this function:



int _base_spe_context_destroy (spe_context_ptr_t spectx)

_base_spe_context_destroy cleans up what is left when an SPE executable has exited. Closes open file handles and unmaps memory areas.

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

Definition at line 418 of file create.c.

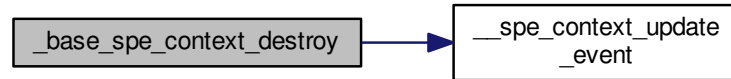
References `__spe_context_update_event()`.

```

419 {
420     int ret = free_spe_context(spe);
421     __spe_context_update_event();
422     return ret;
423 }

```


Here is the call graph for this function:



void _base_spe_context_lock (spe_context_ptr_t spe, enum fd_name fd)

_base_spe_context_lock locks members of the SPE context

Parameters

<i>spectx</i>	Specifies the SPE context
<i>fd</i>	Specifies the file

Definition at line 91 of file create.c.

References spe_context::base_private, and spe_context.base_priv::fd_lock.

Referenced by _base_spe_close_if_open(), and _base_spe_open_if_closed().

```

92 {
93     pthread_mutex_lock(&spe->base_private->fd_lock[fdesc]);
94 }
```

void _base_spe_context_unlock (spe_context_ptr_t spe, enum fd_name fd)

_base_spe_context_unlock unlocks members of the SPE context

Parameters

<i>spectx</i>	Specifies the SPE context
<i>fd</i>	Specifies the file

Definition at line 96 of file create.c.

References spe_context::base_private, and spe_context.base_priv::fd_lock.

Referenced by _base_spe_close_if_open(), and _base_spe_open_if_closed().

```

97 {
98     pthread_mutex_unlock(&spe->base_private->fd_lock[fdesc]);
99 }
```

spe_gang_context_ptr_t _base_spe_gang_context_create (unsigned int flags)

creates the directory in SPUFS that will contain all SPEs that are considered a gang Note: I would like to generalize this to a "group" or "set" Additional attributes maintained at the group level should be used to define scheduling constraints such "temporal" (e.g., scheduled all at the same time, i.e., a gang) "topology" (e.g., "closeness" of SPEs for optimal communication)

Definition at line 376 of file create.c.

References spe_gang_context::base_private, DEBUG_PRINTF, and spe_gang_context.base_priv::gangname.

```

377 {
378     char pathname[256];
379     struct spe_gang_context_base_priv *pgctx = NULL;
380     struct spe_gang_context *gctx = NULL;
381 }
```

```

382     gctx = malloc(sizeof(*gctx));
383     if (!gctx) {
384         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
385         return NULL;
386     }
387     memset(gctx, 0, sizeof(*gctx));
388
389     pgctx = malloc(sizeof(*pgctx));
390     if (!pgctx) {
391         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
392         free(gctx);
393         return NULL;
394     }
395     memset(pgctx, 0, sizeof(*pgctx));
396
397     gctx->base_private = pgctx;
398
399     sprintf(gctx->base_private->gangname, "gang-%i-%lu", getpid(),
400             (unsigned long)gctx);
401     sprintf(pathname, "/spu/%s", gctx->base_private->gangname);
402
403     gctx->base_private->fd.gang_dir = spu_create(pathname, SPU_CREATE_GANG,
404         S_IRUSR | S_IWUSR | S_IXUSR);
405
406     if (gctx->base_private->fd.gang_dir < 0) {
407         DEBUG_PRINTF("ERROR: Could not create Gang %s\n", pathname);
408         free_spe_gang_context(gctx);
409         errno = EFAULT;
410         return NULL;
411     }
412
413     gctx->base_private->flags = flags;
414
415     return gctx;
416 }

```

int _base_spe_gang_context_destroy (spe_gang_context_ptr_t gctx)

_base_spe_gang_context_destroy destroys a gang context and frees associated resources

Parameters

<i>gctx</i>	Specifies the SPE gang context
-------------	--------------------------------

Definition at line 427 of file create.c.

```

428 {
429     return free_spe_gang_context(gctx);
430 }

```

int _base_spe_open_if_closed (struct spe_context * spe, enum fd_name fdesc, int locked)

Definition at line 101 of file create.c.

References _base_spe_context_lock(), _base_spe_context_unlock(), spe_context::base_private, spe_context->base_priv::fd_spe_dir, fd_attr::mode, fd_attr::name, spe_context_base_priv::spe_fds_array, and spe_context->base_priv::spe_fds_refcount.

Referenced by __base_spe_event_source_acquire(), _base_spe_in_mbox_status(), _base_spe_in_mbox_write(), _base_spe_mssync_start(), _base_spe_mssync_status(), _base_spe_out_intr_mbox_read(), _base_spe_out_intr_mbox_status(), _base_spe_out_mbox_read(), _base_spe_out_mbox_status(), and _base_spe_signal_write().

```

102 {
103     if (!locked)
104         _base_spe_context_lock(spe, fdesc);
105
106     /* already open? */
107     if (spe->base_private->spe_fds_array[fdesc] != -1) {
108         spe->base_private->spe_fds_refcount[fdesc]++;
109     } else {
110         spe->base_private->spe_fds_array[fdesc] =
111             openat(spe->base_private->fd_spe_dir,
112                 spe_fd_attr[fdesc].name,

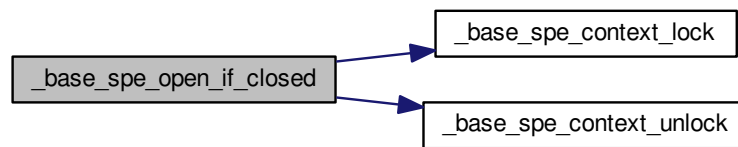
```

```

113         spe_fd_attr[fdesc].mode);
114
115         if (spe->base_private->spe_fds_array[(int)fdesc] > 0)
116             spe->base_private->spe_fds_refcount[(int)fdesc]++;
117     }
118
119     if (!locked)
120         _base_spe_context_unlock(spe, fdesc);
121
122     return spe->base_private->spe_fds_array[(int)fdesc];
123 }

```

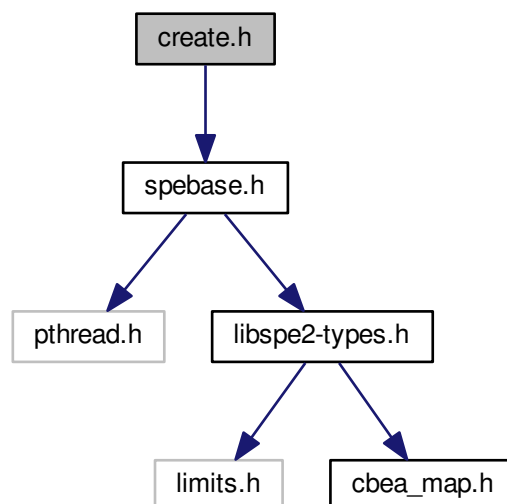
Here is the call graph for this function:



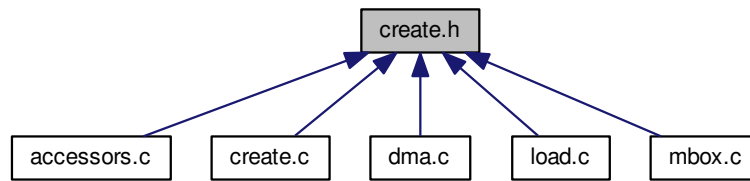
3.4 create.h File Reference

```
#include "spebase.h"
```

Include dependency graph for create.h:



This graph shows which files directly or indirectly include this file:



Functions

- int `_base_spe_open_if_closed` (struct `spe_context` *spe, enum `fd_name` fdesc, int locked)
- void `_base_spe_close_if_open` (struct `spe_context` *spe, enum `fd_name` fdesc)

3.4.1 Function Documentation

void `_base_spe_close_if_open` (struct `spe_context` * *spe*, enum `fd_name` *fdesc*)

Definition at line 125 of file `create.c`.

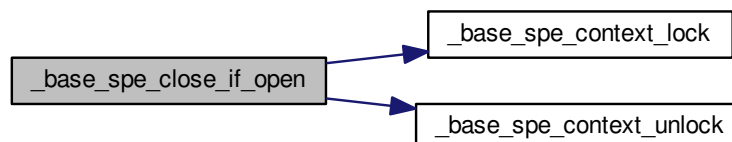
References `_base_spe_context_lock()`, `_base_spe_context_unlock()`, `spe_context::base_private`, `spe_context::base_priv::spe_fds_array`, and `spe_context::base_priv::spe_fds_refcount`.

Referenced by `_base_spe_event_source_release()`, and `_base_spe_signal_write()`.

```

126 {
127     _base_spe_context_lock(spe, fdesc);
128
129     if (spe->base_private->spe_fds_array[(int)fdesc] != -1 &&
130         spe->base_private->spe_fds_refcount[(int)fdesc] == 1) {
131
132         spe->base_private->spe_fds_refcount[(int)fdesc]--;
133         close(spe->base_private->spe_fds_array[(int)fdesc]);
134
135         spe->base_private->spe_fds_array[(int)fdesc] = -1;
136     } else if (spe->base_private->spe_fds_refcount[(int)fdesc] > 0) {
137         spe->base_private->spe_fds_refcount[(int)fdesc]--;
138     }
139
140     _base_spe_context_unlock(spe, fdesc);
141 }
  
```

Here is the call graph for this function:



int _base_spe_open_if_closed (struct spe_context * spe, enum fd_name fdesc, int locked)

Definition at line 101 of file create.c.

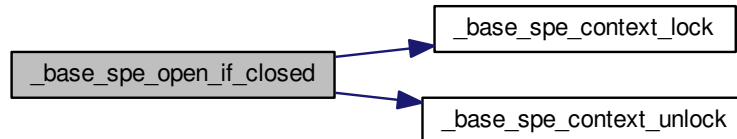
References `_base_spe_context_lock()`, `_base_spe_context_unlock()`, `spe_context::base_private`, `spe_context::base_priv::fd_spe_dir`, `fd_attr::mode`, `fd_attr::name`, `spe_context::base_priv::spe_fds_array`, and `spe_context::base_priv::spe_fds_refcount`.

Referenced by `_base_spe_event_source_acquire()`, `_base_spe_in_mbox_status()`, `_base_spe_in_mbox_write()`, `_base_spe_mssync_start()`, `_base_spe_mssync_status()`, `_base_spe_out_intr_mbox_read()`, `_base_spe_out_intr_mbox_status()`, `_base_spe_out_mbox_read()`, `_base_spe_out_mbox_status()`, and `_base_spe_signal_write()`.

```

102 {
103     if (!locked)
104         _base_spe_context_lock(spe, fdesc);
105
106     /* already open? */
107     if (spe->base_private->spe_fds_array[fdesc] != -1) {
108         spe->base_private->spe_fds_refcount[fdesc]++;
109     } else {
110         spe->base_private->spe_fds_array[fdesc] =
111             openat(spe->base_private->fd_spe_dir,
112                  spe.fd_attr[fdesc].name,
113                  spe.fd_attr[fdesc].mode);
114
115         if (spe->base_private->spe_fds_array[(int)fdesc] > 0)
116             spe->base_private->spe_fds_refcount[(int)fdesc]++;
117     }
118
119     if (!locked)
120         _base_spe_context_unlock(spe, fdesc);
121
122     return spe->base_private->spe_fds_array[(int)fdesc];
123 }
```

Here is the call graph for this function:



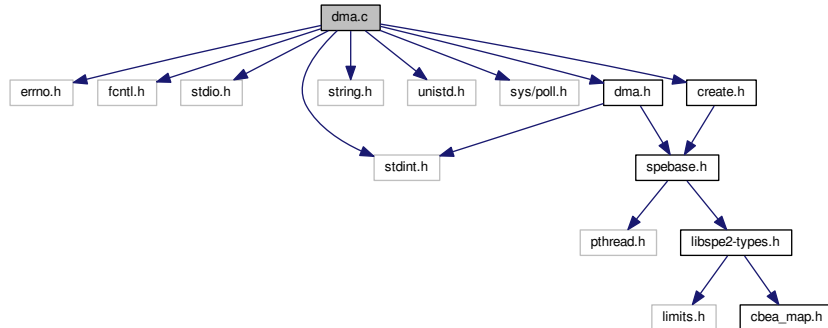
3.5 design.txt File Reference

3.6 dma.c File Reference

```

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <unistd.h>
#include <sys/poll.h>
#include "create.h"
#include "dma.h"
```

Include dependency graph for dma.c:



Functions

- `int _base_spe_mfcio_put (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_putb (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_putf (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_get (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_getb (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_getf (spe_context_ptr_t spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`
- `int _base_spe_mfcio_tag_status_read (spe_context_ptr_t spectx, unsigned int mask, unsigned int behavior, unsigned int *tag_status)`
- `int _base_spe_mssync_start (spe_context_ptr_t spectx)`
- `int _base_spe_mssync_status (spe_context_ptr_t spectx)`

3.6.1 Function Documentation

`int _base_spe_mfcio_get (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)`

The `_base_spe_mfcio_get` function places a get DMA command on the proxy command queue of the SPE thread specified by `speid`. The get command transfers `size` bytes of data starting at the effective address specified by `ea` to the local store address specified by `ls`. The DMA is identified by the tag id specified by `tag` and performed according to the transfer class and replacement class specified by `tid` and `rid` respectively.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.

<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 160 of file dma.c.

References MFC_CMD_GET.

```

167 {
168     return spe_do_mfc_get(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_GET);
169 }
```

int _base_spe_mfcio_getb (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

The `_base_spe_mfcio_getb` function is identical to `_base_spe_mfcio_get` except that it places a getb (get with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all sequence commands with the same tag identifier as this command are locally ordered with respect to all previously issued commands with the same tag group and command queue.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 171 of file dma.c.

References MFC_CMD_GETB.

```

178 {
179     return spe_do_mfc_get(spectx, ls, ea, size, tag, rid, rid, MFC_CMD_GETB);
180 }
```

int _base_spe_mfcio_getf (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

The `_base_spe_mfcio_getf` function is identical to `_base_spe_mfcio_get` except that it places a getf (get with fence) DMA command on the proxy command queue. The fence form ensure that this command is locally ordered with respect to all previously issued commands with the same tag group and command queue.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 182 of file dma.c.

References MFC_CMD_GETF.

```

189 {
190     return spe_do_mfc_get(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_GETF);
191 }
```

int _base_spe_mfcio_put (spe_context_ptr_t *spectx*, unsigned int *ls*, void * *ea*, unsigned int *size*, unsigned int *tag*, unsigned int *tid*, unsigned int *rid*)

The `_base_spe_mfcio_put` function places a put DMA command on the proxy command queue of the SPE thread specified by `speid`. The put command transfers `size` bytes of data starting at the local store address specified by `ls` to the effective address specified by `ea`. The DMA is identified by the tag id specified by `tag` and performed according transfer class and replacement class specified by `tid` and `rid` respectively.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 126 of file dma.c.

References MFC_CMD_PUT.

```

133 {
134     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUT);
135 }
```

int _base_spe_mfcio_putb (spe_context_ptr_t *spectx*, unsigned int *ls*, void * *ea*, unsigned int *size*, unsigned int *tag*, unsigned int *tid*, unsigned int *rid*)

The `_base_spe_mfcio_putb` function is identical to `_base_spe_mfcio_put` except that it places a putb (put with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all sequence commands with the same tag identifier as this command are locally ordered with respect to all previously issued commands with the same tag group and command queue.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 137 of file dma.c.

References MFC_CMD_PUTB.

```

144 {
145     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUTB);
146 }
```

int _base_spe_mfcio_putf (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

The _base_spe_mfcio_putf function is identical to _base_spe_mfcio_put except that it places a putf (put with fence) DMA command on the proxy command queue. The fence form ensures that this command is locally ordered with respect to all previously issued commands with the same tag group and command queue.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 148 of file dma.c.

References MFC_CMD_PUTF.

```

155 {
156     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUTF);
157 }
```

int _base_spe_mfcio_tag_status_read (spe_context_ptr_t spectx, unsigned int mask, unsigned int behavior, unsigned int * tag_status)

_base_spe_mfcio_tag_status_read

No Idea

Definition at line 307 of file dma.c.

References spe_context_base_priv::active_tagmask, spe_context::base_private, spe_context_base_priv::flags, SPE_MAP_PS, SPE_TAG_ALL, SPE_TAG_ANY, and SPE_TAG_IMMEDIATE.

```

308 {
309     if ( mask != 0 ) {
310         if (!(spectx->base_private->flags & SPE_MAP_PS))
311             mask = 0;
312     } else {
313         if ((spectx->base_private->flags & SPE_MAP_PS))
314             mask = spectx->base_private->active_tagmask;
315     }
316
317     if (!tag_status) {
318         errno = EINVAL;
319         return -1;
320     }
321
322     switch (behavior) {
323     case SPE_TAG_ALL:
324         return spe_mfcio_tag_status_read_all(spectx, mask, tag_status);
325     case SPE_TAG_ANY:
326         return spe_mfcio_tag_status_read_any(spectx, mask, tag_status);
327     case SPE_TAG_IMMEDIATE:
328         return spe_mfcio_tag_status_read_immediate(spectx, mask, tag_status);
329     default:
330         errno = EINVAL;
331         return -1;
332     }
333 }

```

int _base_spe_mssync_start (spe_context_ptr_t spectx)

_base_spe_mssync_start starts Multisource Synchronisation

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

Definition at line 335 of file dma.c.

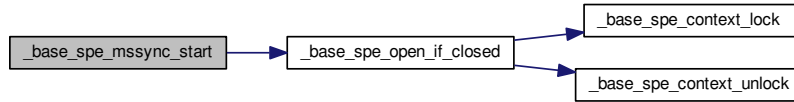
References _base_spe_open_if_closed(), spe_context::base_private, FD_MSS, spe_context_base_priv::flags, spe_mssync_area::MFC_MSSync, spe_context_base_priv::mssync_mmap_base, and SPE_MAP_PS.

```

336 {
337     int ret, fd;
338     unsigned int data = 1; /* Any value can be written here */
339
340     volatile struct spe_mssync_area *mss_area =
341         spectx->base_private->mssync_mmap_base;
342
343     if (spectx->base_private->flags & SPE_MAP_PS) {
344         mss_area->MFC_MSSync = data;
345         return 0;
346     } else {
347         fd = _base_spe_open_if_closed(spectx,
348             FD_MSS, 0);
349         if (fd != -1) {
350             ret = write(fd, &data, sizeof (data));
351             if ((ret < 0) && (errno != EIO)) {
352                 perror("spe_mssync.start: internal error");
353             }
354             return ret < 0 ? -1 : 0;
355         } else
356             return -1;
357     }
358 }

```

Here is the call graph for this function:



int _base_spe_mssync_status (spe_context_ptr_t spectx)

_base_spe_mssync_status retrieves status of Multisource Synchronisation
Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

Definition at line 359 of file dma.c.

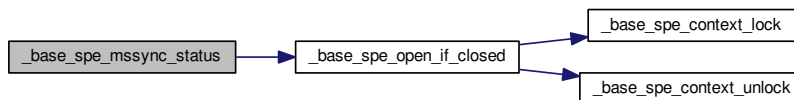
References _base_spe_open_if_closed(), spe_context::base_private, FD_MSS, spe_context_base_priv::flags, spe_mssync_area::MFC_MSSync, spe_context_base_priv::mssync_mmap_base, and SPE_MAP_PS.

```

360 {
361     int ret, fd;
362     unsigned int data;
363
364     volatile struct spe_mssync_area *mss_area =
365         spectx->base_private->mssync_mmap_base;
366
367     if (spectx->base_private->flags & SPE_MAP_PS) {
368         return mss_area->MFC_MSSync;
369     } else {
370         fd = _base_spe_open_if_closed(spectx,
371             FD_MSS, 0);
372         if (fd != -1) {
373             ret = read(fd, &data, sizeof (data));
374             if ((ret < 0) && (errno != EIO)) {
375                 perror("spe_mssync.start: internal error");
376             }
377             return ret < 0 ? -1 : data;
378         } else
379             return -1;
380     }
381 }

```

Here is the call graph for this function:



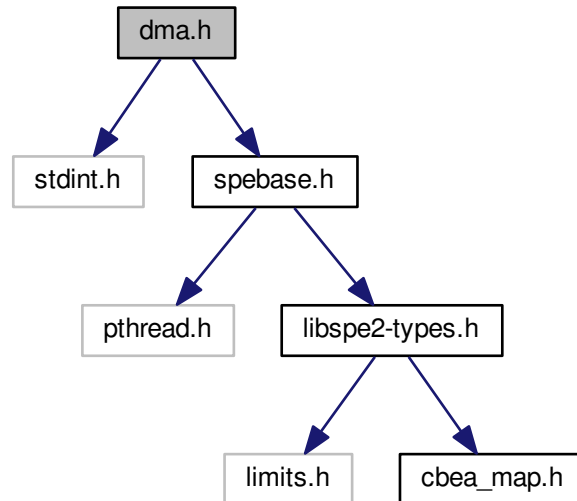
3.7 dma.h File Reference

```

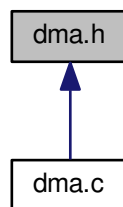
#include <stdint.h>
#include "spebase.h"

```

Include dependency graph for dma.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct `mfc_command_parameter_area`

Enumerations

- enum `mfc_cmd` {
`MFC_CMD_PUT` = 0x20, `MFC_CMD_PUTB` = 0x21, `MFC_CMD_PUTF` = 0x22, `MFC_CMD_GET`
 = 0x40,
`MFC_CMD_GETB` = 0x41, `MFC_CMD_GETF` = 0x42 }

3.7.1 Enumeration Type Documentation

enum mfc_cmd

Enumerator

MFC_CMD_PUT
MFC_CMD_PUTB
MFC_CMD_PUTF
MFC_CMD_GET
MFC_CMD_GETB
MFC_CMD_GETF

Definition at line 37 of file dma.h.

```

37         {
38             MFC_CMD_PUT   = 0x20,
39             MFC_CMD_PUTB  = 0x21,
40             MFC_CMD_PUTF  = 0x22,
41             MFC_CMD_GET   = 0x40,
42             MFC_CMD_GETB  = 0x41,
43             MFC_CMD_GETF  = 0x42,
44     };

```

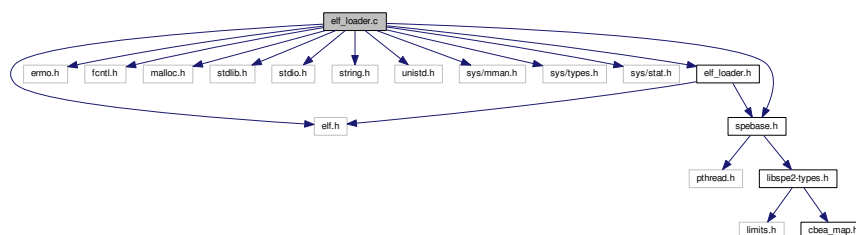
3.8 elf_loader.c File Reference

```

#include <elf.h>
#include <errno.h>
#include <fcntl.h>
#include <malloc.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "elf_loader.h"
#include "spebase.h"

```

Include dependency graph for elf_loader.c:



Macros

- #define **__PRINTF**(fmt, args...) { fprintf(stderr,fmt , ## args); }
- #define **DEBUG_PRINTF**(fmt, args...)
- #define **TAG**

Functions

- `int _base_spe_verify_spe_elf_image (spe_program_handle_t *handle)`
- `int _base_spe_parse_isolated_elf (spe_program_handle_t *handle, uint64_t *addr, uint32_t *size)`
- `int _base_spe_load_spe_elf (spe_program_handle_t *handle, void *ld_buffer, struct spe_ld_info *ld_info)`
- `int _base_spe_toe_ear (spe_program_handle_t *speh)`

3.8.1 Macro Definition Documentation

```
#define __PRINTF( fmt, args... ) { fprintf(stderr,fmt , ## args); }
```

Definition at line 40 of file elf_loader.c.

```
#define DEBUG_PRINTF( fmt, args... )
```

Definition at line 45 of file elf_loader.c.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_count_physical_cpus()`, `_base_spe_count_physical_spes()`, `_base_spe_gang_context_create()`, `_base_spe_handle_library_callback()`, `_base_spe_load_spe_elf()`, `_base_spe_out_mbox_read()`, `_base_spe_parse_isolated_elf()`, `_base_spe_program_load()`, and `_base_spe_program_load_complete()`.

```
#define TAG
```

Definition at line 46 of file elf_loader.c.

3.8.2 Function Documentation

```
int _base_spe_load_spe_elf ( spe_program_handle_t * handle, void * ld_buffer, struct spe_ld_info * ld_info )
```

Definition at line 201 of file elf_loader.c.

References `DEBUG_PRINTF`, `spe_program_handle::elf_image`, and `spe_ld_info::entry`.

Referenced by `_base_spe_program_load()`.

```
202 {
203     Elf32_Ehdr *ehdr;
204     Elf32_Phdr *phdr;
205     Elf32_Phdr *ph, *prev_ph;
206
207     Elf32_Shdr *shdr;
208     Elf32_Shdr *sh;
209
210     Elf32_Off toe_addr = 0;
211     long toe_size = 0;
212
213     char* str_table = 0;
214
215     int num_load_seg = 0;
216     void *elf_start;
217     int ret;
218
219     DEBUG_PRINTF ("load.spe_elf(%p, %p)\n", handle, ld_buffer);
220
221     elf_start = handle->elf_image;
222
223     DEBUG_PRINTF ("load.spe_elf(%p, %p)\n", handle->elf_image, ld_buffer);
224     ehdr = (Elf32_Ehdr *) (handle->elf_image);
225
226     /* Check for a Valid SPE ELF Image (again) */
227     if ((ret=check_spe_elf(ehdr))
228         return ret;
229
230     /* Start processing headers */
231     phdr = (Elf32_Phdr *) ((char *) ehdr + ehdr->e_phoff);
232     shdr = (Elf32_Shdr *) ((char *) ehdr + ehdr->e_shoff);
```

```

233     str.table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;
234
235     /* traverse the sections to locate the toe segment */
236     /* by specification, the toe sections are grouped together in a segment */
237     for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
238     {
239         DEBUG_PRINTF("section name: %s ( start: 0x%04x, size: 0x%04x)\n", str.table+sh
->sh_name, sh->sh_offset, sh->sh_size );
240         if (strcmp(".toe", str.table+sh->sh_name) == 0) {
241             DEBUG_PRINTF("section offset: %d\n", sh->sh_offset);
242             toe_size += sh->sh_size;
243             if ((toe_addr == 0) || (toe_addr > sh->sh_addr))
244                 toe_addr = sh->sh_addr;
245         }
246         /* Disabled : Actually not needed, only good for testing
247         if (strcmp(".bss", str.table+sh->sh_name) == 0) {
248             DEBUG_PRINTF("zeroing .bss section:\n");
249             DEBUG_PRINTF("section offset: 0x%04x\n", sh->sh_offset);
250             DEBUG_PRINTF("section size: 0x%04x\n", sh->sh_size);
251             memset(ld_buffer + sh->sh_offset, 0, sh->sh_size);
252         } */
253
254 #ifdef DEBUG
255         if (strcmp(".note.spu_name", str.table+sh->sh_name) == 0)
256             display_debug_output(elf.start, sh);
257 #endif /*DEBUG*/
258     }
259
260     /*
261     * Load all PT_LOAD segments onto the SPE local store buffer.
262     */
263     DEBUG_PRINTF("Segments: 0x%x\n", ehdr->e_phnum);
264     for (ph = phdr, prev_ph = NULL; ph < &phdr[ehdr->e_phnum]; ++ph) {
265         switch (ph->p_type) {
266             case PT_LOAD:
267                 if (!overlay(ph, prev_ph)) {
268                     if (ph->p_filesz < ph->p_memsz) {
269                         DEBUG_PRINTF("padding loaded image with zeros:\n");
270                         DEBUG_PRINTF("start: 0x%04x\n", ph->p_vaddr + ph->
271 p_filesz);
272                         DEBUG_PRINTF("length: 0x%04x\n", ph->p_memsz - ph->
273 p_filesz);
274                         memset(ld_buffer + ph->p_vaddr + ph->p_filesz, 0, ph->p_memsz - ph
->p_filesz);
275                     }
276                     copy_to_ld_buffer(handle, ld_buffer, ph,
277                                     toe_addr, toe_size);
278                     num_load_seg++;
279                 }
280                 break;
281             case PT_NOTE:
282                 DEBUG_PRINTF("SPE_LOAD found PT_NOTE\n");
283                 break;
284         }
285     }
286     if (num_load_seg == 0)
287     {
288         DEBUG_PRINTF ("no segments to load");
289         errno = EINVAL;
290         return -errno;
291     }
292
293     /* Remember where the code wants to be started */
294     ld.info->entry = ehdr->e_entry;
295     DEBUG_PRINTF ("entry = 0x%x\n", ehdr->e_entry);
296
297     return 0;
298 }

```

int _base_spe_parse_isolated_elf (spe_program_handle_t * handle, uint64_t * addr, uint32_t * size)

Definition at line 111 of file elf_loader.c.

References `DEBUG_PRINTF`, and `spe_program_handle::elf_image`.

```

113 {
114     Elf32_Ehdr *ehdr = (Elf32_Ehdr *)handle->elf_image;

```

```

115     Elf32_Phdr *phdr;
116
117     if (!ehdr) {
118         DEBUG_PRINTF("No ELF image has been loaded\n");
119         errno = EINVAL;
120         return -errno;
121     }
122
123     if (ehdr->e_phentsize != sizeof(*phdr)) {
124         DEBUG_PRINTF("Invalid program header format (phdr size=%d)\n",
125                     ehdr->e_phentsize);
126         errno = EINVAL;
127         return -errno;
128     }
129
130     if (ehdr->e_phnum != 1) {
131         DEBUG_PRINTF("Invalid program header count (%d), expected 1\n",
132                     ehdr->e_phnum);
133         errno = EINVAL;
134         return -errno;
135     }
136
137     phdr = (Elf32_Phdr *) (handle->elf_image + ehdr->e_phoff);
138
139     if (phdr->p_type != PT_LOAD || phdr->p_memsz == 0) {
140         DEBUG_PRINTF("SPE program segment is not loadable (type=%x)\n",
141                     phdr->p_type);
142         errno = EINVAL;
143         return -errno;
144     }
145
146     if (addr)
147         *addr = (uint64_t) (unsigned long)
148                 (handle->elf_image + phdr->p_offset);
149
150     if (size)
151         *size = phdr->p_memsz;
152
153     return 0;
154 }

```

int _base_spe_toe_ea (spe_program_handle_t * speh)

Definition at line 354 of file elf_loader.c.

References `spe_program_handle::elf_image`, and `spe_program_handle::toe_shadow`.

Referenced by `_base_spe_image_open()`.

```

355 {
356     Elf32_Ehdr *ehdr;
357     Elf32_Shdr *shdr, *sh;
358     char *str_table;
359     char **ch;
360     int ret;
361     long toe_size;
362
363     ehdr = (Elf32_Ehdr*) (speh->elf_image);
364     shdr = (Elf32_Shdr*) ((char*) ehdr + ehdr->e_shoff);
365     str_table = (char*) ehdr + shdr[ehdr->e_shstrndx].sh_offset;
366
367     toe_size = 0;
368     for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
369         if (strcmp(".toe", str_table + sh->sh_name) == 0)
370             toe_size += sh->sh_size;
371
372     ret = 0;
373     if (toe_size > 0) {
374         for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
375             if (sh->sh_type == SHT_SYMTAB || sh->sh_type ==
376                 SHT_DYNSYM)
377                 ret = toe_check_syms(ehdr, sh);
378         if (!ret && toe_size != 16) {
379             /* Paranoia */
380             fprintf(stderr, "Unexpected toe size of %ld\n",
381                     toe_size);
382             errno = EINVAL;
383             ret = 1;
384         }
385     }
386 }

```



```

384     }
385 }
386 if (!ret && toe_size) {
387     /*
388      * Allocate toe_shadow, and fill it with elf_image.
389      */
390     speh->toe_shadow = malloc(toe_size);
391     if (speh->toe_shadow) {
392         ch = (char**) speh->toe_shadow;
393         if (sizeof(char*) == 8) {
394             ch[0] = (char*) speh->elf_image;
395             ch[1] = 0;
396         } else {
397             ch[0] = 0;
398             ch[1] = (char*) speh->elf_image;
399             ch[2] = 0;
400             ch[3] = 0;
401         }
402     } else {
403         errno = ENOMEM;
404         ret = 1;
405     }
406 }
407 return ret;
408 }

```

int _base_spe_verify_spe_elf_image (spe_program_handle_t * handle)

verifies integrity of an SPE image

Definition at line 99 of file elf_loader.c.

References spe_program_handle::elf_image.

Referenced by _base_spe_emulated_loader_present(), and _base_spe_image_open().

```

100 {
101     Elf32_Ehdr *ehdr;
102     void *elf_start;
103
104     elf_start = handle->elf_image;
105     ehdr = (Elf32_Ehdr *) (handle->elf_image);
106
107     return check_spe_elf(ehdr);
108 }

```

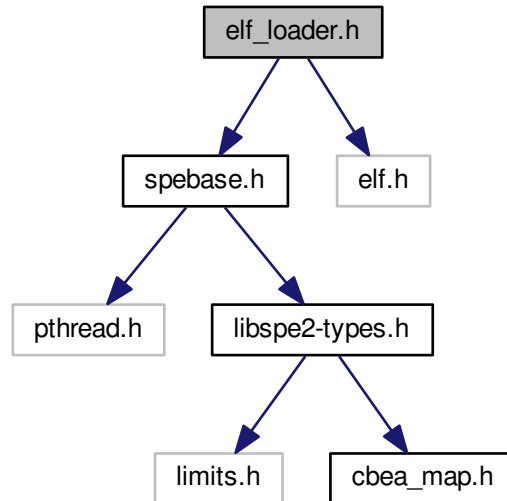
3.9 elf_loader.h File Reference

```

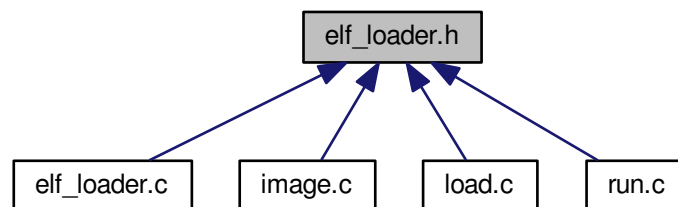
#include "spebase.h"
#include <elf.h>

```

Include dependency graph for elf_loader.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- union `addr64`
- struct `spe_ld_info`

Macros

- `#define LS_SIZE 0x40000 /* 256K (in bytes) */`
- `#define SPE_LDR_PROG_start (LS_SIZE - 512)`
- `#define SPE_LDR_PARAMS_start (LS_SIZE - 128)`

Functions

- `int _base_spe_verify_spe_elf_image (spe_program_handle_t *handle)`
- `int _base_spe_load_spe_elf (spe_program_handle_t *handle, void *ld_buffer, struct spe_ld_info *ld_info)`
- `int _base_spe_parse_isolated_elf (spe_program_handle_t *handle, uint64_t *addr, uint32_t *size)`
- `int _base_spe_toe_ear (spe_program_handle_t *speh)`

3.9.1 Macro Definition Documentation

#define LS_SIZE 0x40000 /* 256K (in bytes) */

Definition at line 23 of file elf_loader.h.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, and `_base_spe_ls_size_get()`.

#define SPE_LDR_PARAMS_start (LS_SIZE - 128)

Definition at line 26 of file elf_loader.h.

#define SPE_LDR_PROG_start (LS_SIZE - 512)

Definition at line 25 of file elf_loader.h.

3.9.2 Function Documentation

int _base_spe_load_spe_elf (spe_program_handle_t * *handle*, void * *ld_buffer*, struct spe_ld_info * *ld_info*)

Definition at line 201 of file elf_loader.c.

References `DEBUG_PRINTF`, `spe_program_handle::elf_image`, and `spe_ld_info::entry`.

Referenced by `_base_spe_program_load()`.

```

202 {
203     Elf32_Ehdr *ehdr;
204     Elf32_Phdr *phdr;
205     Elf32_Phdr *ph, *prev_ph;
206
207     Elf32_Shdr *shdr;
208     Elf32_Shdr *sh;
209
210     Elf32_Off  toe_addr = 0;
211     long      toe_size = 0;
212
213     char* str_table = 0;
214
215     int num_load_seg = 0;
216     void *elf_start;
217     int ret;
218
219     DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle, ld_buffer);
220
221     elf_start = handle->elf_image;
222
223     DEBUG_PRINTF ("load_spe_elf(%p, %p)\n", handle->elf_image, ld_buffer);
224     ehdr = (Elf32_Ehdr *) (handle->elf_image);
225
226     /* Check for a Valid SPE ELF Image (again) */
227     if ((ret=check_spe_elf(ehdr)))
228         return ret;
229
230     /* Start processing headers */
231     phdr = (Elf32_Phdr *) ((char *) ehdr + ehdr->e_phoff);
232     shdr = (Elf32_Shdr *) ((char *) ehdr + ehdr->e_shoff);
233     str_table = (char*)ehdr + shdr[ehdr->e_shstrndx].sh_offset;
234
235     /* traverse the sections to locate the toe segment */
236     /* by specification, the toe sections are grouped together in a segment */

```

```

237     for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
238     {
239         DEBUG_PRINTF("section name: %s ( start: 0x%04x, size: 0x%04x)\n", str_table+sh
->sh_name, sh->sh_offset, sh->sh_size );
240         if (strcmp(".toe", str_table+sh->sh_name) == 0) {
241             DEBUG_PRINTF("section offset: %d\n", sh->sh_offset);
242             toe_size += sh->sh_size;
243             if ((toe_addr == 0) || (toe_addr > sh->sh_addr))
244                 toe_addr = sh->sh_addr;
245         }
246         /* Disabled : Actually not needed, only good for testing
247         if (strcmp(".bss", str_table+sh->sh_name) == 0) {
248             DEBUG_PRINTF("zeroing .bss section:\n");
249             DEBUG_PRINTF("section offset: 0x%04x\n", sh->sh_offset);
250             DEBUG_PRINTF("section size: 0x%04x\n", sh->sh_size);
251             memset(ld.buffer + sh->sh_offset, 0, sh->sh_size);
252         } */
253
254 #ifdef DEBUG
255         if (strcmp(".note.spu_name", str_table+sh->sh_name) == 0)
256             display_debug_output(elf.start, sh);
257 #endif /*DEBUG*/
258     }
259
260     /*
261     * Load all PT_LOAD segments onto the SPE local store buffer.
262     */
263     DEBUG_PRINTF("Segments: 0x%x\n", ehdr->e_phnum);
264     for (ph = phdr, prev_ph = NULL; ph < &phdr[ehdr->e_phnum]; ++ph) {
265         switch (ph->p_type) {
266             case PT_LOAD:
267                 if (!overlay(ph, prev_ph)) {
268                     if (ph->p_filesz < ph->p_memsz) {
269                         DEBUG_PRINTF("padding loaded image with zeros:\n");
270                         DEBUG_PRINTF("start: 0x%04x\n", ph->p_vaddr + ph->
p_filesz);
271                         DEBUG_PRINTF("length: 0x%04x\n", ph->p_memsz - ph->
p_filesz);
272                         memset(ld.buffer + ph->p_vaddr + ph->p_filesz, 0, ph->p_memsz - ph
->p_filesz);
273                     }
274                     copy_to_ld_buffer(handle, ld.buffer, ph,
275                                     toe_addr, toe_size);
276                     num_load_seg++;
277                 }
278                 break;
279             case PT_NOTE:
280                 DEBUG_PRINTF("SPE_LOAD found PT_NOTE\n");
281                 break;
282         }
283     }
284     if (num_load_seg == 0)
285     {
286         DEBUG_PRINTF ("no segments to load");
287         errno = EINVAL;
288         return -errno;
289     }
290
291     /* Remember where the code wants to be started */
292     ld.info->entry = ehdr->e_entry;
293     DEBUG_PRINTF ("entry = 0x%x\n", ehdr->e_entry);
294
295     return 0;
296
297 }

```

int _base_spe_parse_isolated_elf (spe_program_handle_t * handle, uint64_t * addr, uint32_t * size)

Definition at line 111 of file elf_loader.c.

References `DEBUG_PRINTF`, and `spe_program_handle::elf_image`.

```

113 {
114     Elf32_Ehdr *ehdr = (Elf32_Ehdr *)handle->elf_image;
115     Elf32_Phdr *phdr;
116
117     if (!ehdr) {
118         DEBUG_PRINTF("No ELF image has been loaded\n");

```

```

119         errno = EINVAL;
120         return -errno;
121     }
122
123     if (ehdr->e_phentsize != sizeof(*phdr)) {
124         DEBUG.PRINTF("Invalid program header format (phdr size=%d)\n",
125                     ehdr->e_phentsize);
126         errno = EINVAL;
127         return -errno;
128     }
129
130     if (ehdr->e_phnum != 1) {
131         DEBUG.PRINTF("Invalid program header count (%d), expected 1\n",
132                     ehdr->e_phnum);
133         errno = EINVAL;
134         return -errno;
135     }
136
137     phdr = (Elf32_Phdr *) (handle->elf_image + ehdr->e_phoff);
138
139     if (phdr->p_type != PT_LOAD || phdr->p_memsz == 0) {
140         DEBUG.PRINTF("SPE program segment is not loadable (type=%x)\n",
141                     phdr->p_type);
142         errno = EINVAL;
143         return -errno;
144     }
145
146     if (addr)
147         *addr = (uint64_t) (unsigned long)
148                 (handle->elf_image + phdr->p_offset);
149
150     if (size)
151         *size = phdr->p_memsz;
152
153     return 0;
154 }

```

int _base_spe_toe_ea (spe_program_handle_t * speh)

Definition at line 354 of file elf_loader.c.

References `spe_program_handle::elf_image`, and `spe_program_handle::toe_shadow`.

Referenced by `_base_spe_image_open()`.

```

355 {
356     Elf32_Ehdr *ehdr;
357     Elf32_Shdr *shdr, *sh;
358     char *str_table;
359     char **ch;
360     int ret;
361     long toe_size;
362
363     ehdr = (Elf32_Ehdr *) (speh->elf_image);
364     shdr = (Elf32_Shdr *) ((char *) ehdr + ehdr->e_shoff);
365     str_table = (char *) ehdr + shdr[ehdr->e_shstrndx].sh_offset;
366
367     toe_size = 0;
368     for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
369         if (strcmp(".toe", str_table + sh->sh_name) == 0)
370             toe_size += sh->sh_size;
371
372     ret = 0;
373     if (toe_size > 0) {
374         for (sh = shdr; sh < &shdr[ehdr->e_shnum]; ++sh)
375             if (sh->sh_type == SHT_SYMTAB || sh->sh_type ==
376                 SHT_DYNSYM)
377                 ret = toe_check_syms(ehdr, sh);
378         if (!ret && toe_size != 16) {
379             /* Paranoia */
380             fprintf(stderr, "Unexpected toe size of %ld\n",
381                     toe_size);
382             errno = EINVAL;
383             ret = 1;
384         }
385     }
386     if (!ret && toe_size) {
387         /*

```

```

388         * Allocate toe_shadow, and fill it with elf_image.
389         */
390     speh->toe_shadow = malloc(toe_size);
391     if (speh->toe_shadow) {
392         ch = (char**) speh->toe_shadow;
393         if (sizeof(char*) == 8) {
394             ch[0] = (char*) speh->elf_image;
395             ch[1] = 0;
396         } else {
397             ch[0] = 0;
398             ch[1] = (char*) speh->elf_image;
399             ch[2] = 0;
400             ch[3] = 0;
401         }
402     } else {
403         errno = ENOMEM;
404         ret = 1;
405     }
406 }
407 return ret;
408 }

```

int _base_spe_verify_spe_elf_image (spe_program_handle_t * handle)

verifies integrity of an SPE image

Definition at line 99 of file elf_loader.c.

References `spe_program_handle::elf_image`.

Referenced by `_base_spe_emulated_loader_present()`, and `_base_spe_image_open()`.

```

100 {
101     Elf32_Ehdr *ehdr;
102     void *elf_start;
103
104     elf_start = handle->elf_image;
105     ehdr = (Elf32_Ehdr *) (handle->elf_image);
106
107     return check_spe_elf(ehdr);
108 }

```

3.10 handler_utils.h File Reference

Data Structures

- struct `spe_reg128`

Macros

- #define `LS_SIZE` 0x40000 /* 256K (in bytes) */
- #define `LS_ADDR_MASK` (LS_SIZE - 1)
- #define `__PRINTF`(fmt, args...) { fprintf(stderr,fmt , ## args); }
- #define `DEBUG_PRINTF`(fmt, args...)
- #define `LS_ARG_ADDR`(_index) (&((struct `spe_reg128` *) ((char *) ls + ls_args))[_index])
- #define `DECL_RET`() struct `spe_reg128` *ret = `LS_ARG_ADDR`(0)
- #define `GET_LS_PTR`(_off) (void *) ((char *) ls + ((_off) & LS_ADDR_MASK))
- #define `GET_LS_PTR_NULL`(_off) ((_off) ? `GET_LS_PTR`(_off) : NULL)
- #define `DECL_0_ARGS`() unsigned int ls_args = (opdata & 0xfffff)
- #define `DECL_1_ARGS`()
- #define `DECL_2_ARGS`()
- #define `DECL_3_ARGS`()
- #define `DECL_4_ARGS`()
- #define `DECL_5_ARGS`()
- #define `DECL_6_ARGS`()
- #define `PUT_LS_RC`(_a, _b, _c, _d)

3.10.1 Macro Definition Documentation

#define __PRINTF(*fmt*, *args...*) { fprintf(stderr,fmt , ## args); }

Definition at line 32 of file handler_utils.h.

#define DEBUG_PRINTF(*fmt*, *args...*)

Definition at line 36 of file handler_utils.h.

#define DECL_0_ARGS() unsigned int ls_args = (opdata & 0xfffff)

Definition at line 51 of file handler_utils.h.

#define DECL_1_ARGS()

Value:

```
DECL_0_ARGS();
    struct spe_reg128 *arg0 = LS_ARG_ADDR(0) \
```

Definition at line 54 of file handler_utils.h.

#define DECL_2_ARGS()

Value:

```
DECL_1_ARGS();
    struct spe_reg128 *arg1 = LS_ARG_ADDR(1) \
```

Definition at line 58 of file handler_utils.h.

#define DECL_3_ARGS()

Value:

```
DECL_2_ARGS();
    struct spe_reg128 *arg2 = LS_ARG_ADDR(2) \
```

Definition at line 62 of file handler_utils.h.

#define DECL_4_ARGS()

Value:

```
DECL_3_ARGS();
    struct spe_reg128 *arg3 = LS_ARG_ADDR(3) \
```

Definition at line 66 of file handler_utils.h.

#define DECL_5_ARGS()

Value:

```
DECL_4_ARGS();
    struct spe_reg128 *arg4 = LS_ARG_ADDR(4) \
```

Definition at line 70 of file handler_utils.h.

#define DECL_6_ARGS()

Value:

```
DECL_5_ARGS();
    struct spe_reg128 *arg5 = LS_ARG_ADDR(5) \
```

Definition at line 74 of file handler_utils.h.

#define DECL_RET() struct spe_reg128 *ret = LS_ARG_ADDR(0)

Definition at line 42 of file handler_utils.h.

#define GET_LS_PTR(_off) (void *) ((char *) ls + ((_off) & LS_ADDR_MASK))

Definition at line 45 of file handler_utils.h.

#define GET_LS_PTR_NULL(_off) ((_off) ? GET_LS_PTR(_off) : NULL)

Definition at line 48 of file handler_utils.h.

#define LS_ADDR_MASK (LS_SIZE - 1)

Definition at line 29 of file handler_utils.h.

#define LS_ARG_ADDR(_index) (&((struct spe_reg128 *) ((char *) ls + ls.args))[_index])

Definition at line 39 of file handler_utils.h.

#define LS_SIZE 0x40000 /* 256K (in bytes) */

Definition at line 28 of file handler_utils.h.

#define PUT_LS_RC(_a, _b, _c, _d)

Value:

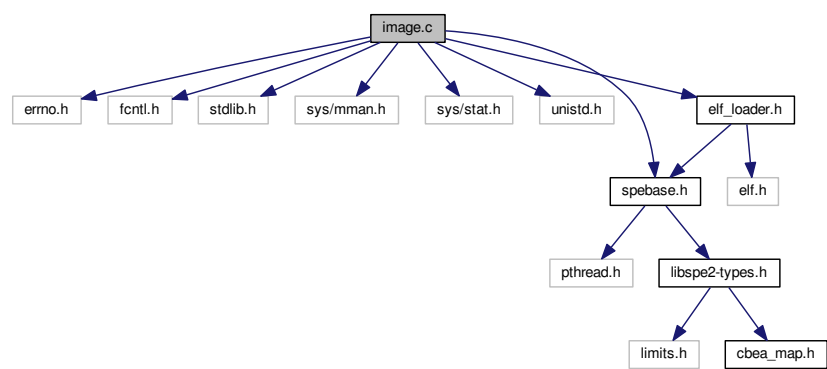
```
ret->slot[0] = (unsigned int) (_a);
ret->slot[1] = (unsigned int) (_b);
ret->slot[2] = (unsigned int) (_c);
ret->slot[3] = (unsigned int) (_d);
__asm__ __volatile__ ("sync" : : : "memory") \
```

Definition at line 78 of file handler_utils.h.

3.11 image.c File Reference

```
#include <errno.h>
#include <fcntl.h>
#include <stdlib.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#include "elf_loader.h"
#include "spebase.h"
```


Include dependency graph for image.c:



Data Structures

- struct `image_handle`

Functions

- `spe_program_handle_t * _base_spe_image_open` (const char *filename)
- `int _base_spe_image_close` (spe_program_handle_t *handle)

3.11.1 Function Documentation

`int _base_spe_image_close (spe_program_handle_t * handle)`

`_base_spe_image_close` unmaps an SPE ELF object that was previously mapped using `spe_open_image`.

Parameters

<i>handle</i>	handle to open file
---------------	---------------------

Return values

0	On success, <code>spe_close_image</code> returns 0.
-1	On failure, -1 is returned and <code>errno</code> is set appropriately. Possible values for <code>errno</code> : EINVAL From <code>spe_close_image</code> , this indicates that the file, specified by filename, was not previously mapped by a call to <code>spe_open_image</code> .

Definition at line 96 of file `image.c`.

References `spe_program_handle::elf_image`, `image_handle::map_size`, `image_handle::speh`, and `spe_program_handle::toe_shadow`.

```
97 {
98     int ret = 0;
99     struct image_handle *ih;
100
101     if (!handle) {
102         errno = EINVAL;
103         return -1;
104     }
105
106     ih = (struct image_handle *)handle;
107 }
```

```

108     if (!ih->speh.elf_image || !ih->map_size) {
109         errno = EINVAL;
110         return -1;
111     }
112
113     if (ih->speh.toe_shadow)
114         free(ih->speh.toe_shadow);
115
116     ret = munmap(ih->speh.elf_image, ih->map_size);
117     free(handle);
118
119     return ret;
120 }

```

spe_program_handle_t* _base_spe_image_open (const char *filename)

_base_spe_image_open maps an SPE ELF executable indicated by filename into system memory and returns the mapped address appropriate for use by the spe_create_thread API. It is often more convenient/appropriate to use the loading methodologies where SPE ELF objects are converted to PPE static or shared libraries with symbols which point to the SPE ELF objects after these special libraries are loaded. These libraries are then linked with the associated PPE code to provide a direct symbol reference to the SPE ELF object. The symbols in this scheme are equivalent to the address returned from the spe_open_image function. SPE ELF objects loaded using this function are not shared with other processes, but SPE ELF objects loaded using the other scheme, mentioned above, can be shared if so desired.

Parameters

<i>filename</i>	Specifies the filename of an SPE ELF executable to be loaded and mapped into system memory.
-----------------	---

Returns

On success, spe_open_image returns the address at which the specified SPE ELF object has been mapped. On failure, NULL is returned and errno is set appropriately.

Possible values for errno include:

EACCES The calling process does not have permission to access the specified file.

EFAULT The filename parameter points to an address that was not contained in the calling process's address space.

A number of other errno values could be returned by the open(2), fstat(2), mmap(2), munmap(2), or close(2) system calls which may be utilized by the spe_open_image or spe_close_image functions.

See Also

spe_create_thread

Definition at line 37 of file image.c.

References _base_spe_toe_ea(), _base_spe_verify_spe_elf_image(), spe_program_handle::elf_image, spe_program_handle::handle_size, image_handle::map_size, image_handle::speh, and spe_program_handle::toe_shadow.

```

38 {
39     /* allocate an extra integer in the spe handle to keep the mapped size information */
40     struct image_handle *ret;
41     int binfd = -1, fstat;
42     struct stat statbuf;
43     size_t ps = getpagesize ();
44
45     ret = malloc(sizeof(struct image_handle));
46     if (!ret)
47         return NULL;
48
49     ret->speh.handle_size = sizeof(spe_program_handle_t);
50     ret->speh.toe_shadow = NULL;
51 }

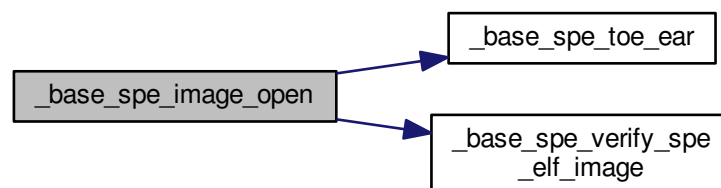
```

```

52     binfd = open(filename, O_RDONLY);
53     if (binfd < 0)
54         goto ret_err;
55
56     f_stat = fstat(binfd, &statbuf);
57     if (f_stat < 0)
58         goto ret_err;
59
60     /* Sanity: is it executable ?
61     */
62     if (!(statbuf.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH))) {
63         errno=EACCES;
64         goto ret_err;
65     }
66
67     /* now store the size at the extra allocated space */
68     ret->map_size = (statbuf.st_size + ps - 1) & ~(ps - 1);
69
70     ret->speh.elf_image = mmap(NULL, ret->map_size,
71                               PROT_WRITE | PROT_READ,
72                               MAP_PRIVATE, binfd, 0);
73     if (ret->speh.elf_image == MAP_FAILED)
74         goto ret_err;
75
76     /*Verify that this is a valid SPE ELF object*/
77     if ((_base_spe_verify_spe_elf_image((
78         spe_program_handle_t *)ret)))
79         goto ret_err;
80
81     if (_base_spe_toe_ear(&ret->speh))
82         goto ret_err;
83
84     /* ok */
85     close(binfd);
86     return (spe_program_handle_t *)ret;
87
88 /* err & cleanup */
89 ret_err:
90     if (binfd >= 0)
91         close(binfd);
92
93     free(ret);
94     return NULL;
95 }

```

Here is the call graph for this function:



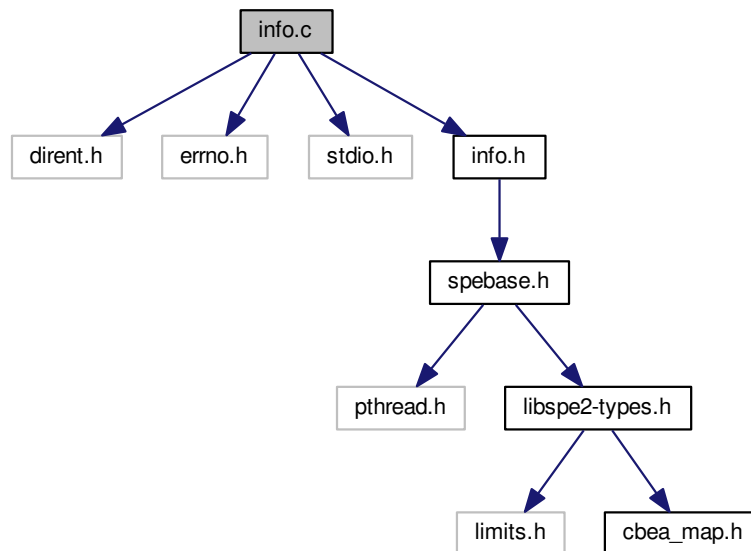
3.12 info.c File Reference

```

#include <dirent.h>
#include <errno.h>
#include <stdio.h>
#include "info.h"

```

Include dependency graph for info.c:



Functions

- `int _base_spe_count_physical_cpus (int cpu_node)`
- `int _base_spe_count_usable_spes (int cpu_node)`
- `int _base_spe_count_physical_spes (int cpu_node)`
- `int _base_spe_cpu_info_get (int info_requested, int cpu_node)`

3.12.1 Function Documentation

int _base_spe_count_physical_cpus (int *cpu_node*)

Definition at line 30 of file info.c.

References `DEBUG_PRINTF`, and `THREADS_PER_BE`.

Referenced by `_base_spe_count_physical_spes()`, and `_base_spe_cpu_info_get()`.

```

31 {
32     const char    *buff = "/sys/devices/system/cpu";
33     DIR          *dirp;
34     int ret = -2;
35     struct dirent *dp;
36
37     DEBUG_PRINTF ("spe_count_physical_cpus()\n");
38
39     // make sure, cpu_node is in the correct range
40     if (cpu_node != -1) {
41         errno = EINVAL;
42         return -1;
43     }
44
45     // Count number of CPUs in /sys/devices/system/cpu
46     if ((dirp=opendir(buff))!=NULL) {
47         fprintf(stderr,"Error opening %s ",buff);
48         perror("dirlist");
49         errno = EINVAL;

```

```

50         return -1;
51     }
52     while((dptr=readdir(dirp))) {
53         ret++;
54     }
55     closedir(dirp);
56     return ret/THREADS_PER_BE;
57 }

```

int _base_spe_count_physical_spes (int cpu_node)

Definition at line 71 of file info.c.

References `_base_spe_count_physical_cpus()`, and `DEBUG_PRINTF`.

Referenced by `_base_spe_count_usable_spes()`, and `_base_spe_cpu_info_get()`.

```

72 {
73     const char    *buff = "/sys/devices/system/spu";
74     DIR            *dirp;
75     int ret = -2;
76     struct dirent *dptr;
77     int no_of_bes;
78
79     DEBUG_PRINTF ("spe_count_physical_spes()\n");
80
81     // make sure, cpu_node is in the correct range
82     no_of_bes = _base_spe_count_physical_cpus(-1);
83     if (cpu_node < -1 || cpu_node >= no_of_bes ) {
84         errno = EINVAL;
85         return -1;
86     }
87
88     // Count number of SPUs in /sys/devices/system/spu
89     if((dirp=opendir(buff))==NULL) {
90         fprintf(stderr,"Error opening %s ",buff);
91         perror("dirlist");
92         errno = EINVAL;
93         return -1;
94     }
95     while((dptr=readdir(dirp))) {
96         ret++;
97     }
98     closedir(dirp);
99
100     if(cpu_node != -1) ret /= no_of_bes; // FIXME
101     return ret;
102 }

```

Here is the call graph for this function:



int _base_spe_count_usable_spes (int cpu_node)

Definition at line 62 of file info.c.

References `_base_spe_count_physical_spes()`.

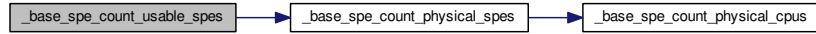
Referenced by `_base_spe_cpu_info_get()`.

```

63 {
64     return _base_spe_count_physical_spes(cpu_node); // FIXME
65 }

```

Here is the call graph for this function:



int _base_spe_cpu_info_get (int *info_requested*, int *cpu_node*)

`_base_spe_info_get`

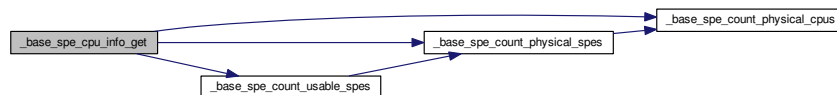
Definition at line 105 of file `info.c`.

References `_base_spe_count_physical_cpus()`, `_base_spe_count_physical_spes()`, `_base_spe_count_usable_spes()`, `SPE_COUNT_PHYSICAL_CPU_NODES`, `SPE_COUNT_PHYSICAL_SPES`, and `SPE_COUNT_USABLE_SPES`.

```

105                                     {
106     int ret = 0;
107     errno = 0;
108
109     switch (info_requested) {
110     case SPE_COUNT_PHYSICAL_CPU_NODES:
111         ret = _base_spe_count_physical_cpus (cpu_node);
112         break;
113     case SPE_COUNT_PHYSICAL_SPES:
114         ret = _base_spe_count_physical_spes (cpu_node);
115         break;
116     case SPE_COUNT_USABLE_SPES:
117         ret = _base_spe_count_usable_spes (cpu_node);
118         break;
119     default:
120         errno = EINVAL;
121         ret = -1;
122     }
123     return ret;
124 }
  
```

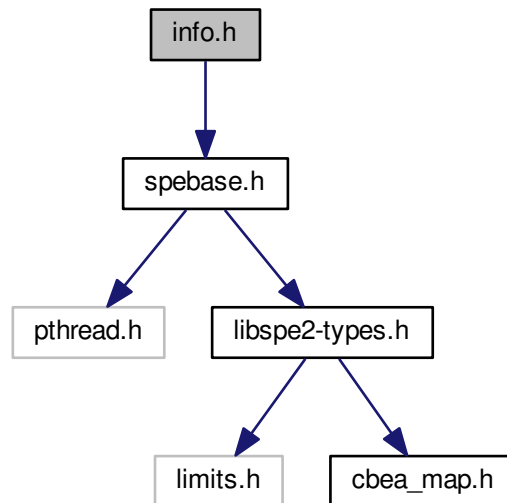
Here is the call graph for this function:



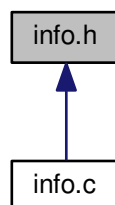
3.13 info.h File Reference

```
#include "spebase.h"
```

Include dependency graph for info.h:



This graph shows which files directly or indirectly include this file:



Macros

- `#define` [THREADS_PER_BE](#) 2

Functions

- `int` [_base_spe_count_physical_cpus](#) (`int` `cpu_node`)
- `int` [_base_spe_count_physical_spes](#) (`int` `cpu_node`)
- `int` [_base_spe_count_usable_spes](#) (`int` `cpu_node`)

3.13.1 Macro Definition Documentation

#define THREADS_PER_BE 2

Definition at line 25 of file info.h.

Referenced by `_base_spe_count_physical_cpus()`.

3.13.2 Function Documentation

int _base_spe_count_physical_cpus (int *cpu_node*)

Definition at line 30 of file info.c.

References `DEBUG_PRINTF`, and `THREADS_PER_BE`.

Referenced by `_base_spe_count_physical_spes()`, and `_base_spe_cpu_info_get()`.

```

31 {
32     const char    *buff = "/sys/devices/system/cpu";
33     DIR            *dirp;
34     int ret = -2;
35     struct dirent *dptr;
36
37     DEBUG_PRINTF ("spe_count_physical_cpus()\n");
38
39     // make sure, cpu_node is in the correct range
40     if (cpu_node != -1) {
41         errno = EINVAL;
42         return -1;
43     }
44
45     // Count number of CPUs in /sys/devices/system/cpu
46     if ((dirp=opendir(buff))!=NULL) {
47         fprintf(stderr,"Error opening %s ",buff);
48         perror("dirlist");
49         errno = EINVAL;
50         return -1;
51     }
52     while((dptr=readdir(dirp))) {
53         ret++;
54     }
55     closedir(dirp);
56     return ret/THREADS_PER_BE;
57 }

```

int _base_spe_count_physical_spes (int *cpu_node*)

Definition at line 71 of file info.c.

References `_base_spe_count_physical_cpus()`, and `DEBUG_PRINTF`.

Referenced by `_base_spe_count_usable_spes()`, and `_base_spe_cpu_info_get()`.

```

72 {
73     const char    *buff = "/sys/devices/system/spu";
74     DIR            *dirp;
75     int ret = -2;
76     struct dirent *dptr;
77     int no_of_bes;
78
79     DEBUG_PRINTF ("spe_count_physical_spes()\n");
80
81     // make sure, cpu_node is in the correct range
82     no_of_bes = _base_spe_count_physical_cpus(-1);
83     if (cpu_node < -1 || cpu_node >= no_of_bes) {
84         errno = EINVAL;
85         return -1;
86     }
87
88     // Count number of SPUs in /sys/devices/system/spu
89     if ((dirp=opendir(buff))!=NULL) {
90         fprintf(stderr,"Error opening %s ",buff);
91         perror("dirlist");
92         errno = EINVAL;
93         return -1;

```



```

94     }
95     while((dptr=readdir(dirp))) {
96         ret++;
97     }
98     closedir(dirp);
99
100     if(cpu_node != -1) ret /= no_of_bes; // FIXME
101     return ret;
102 }

```

Here is the call graph for this function:



int _base_spe_count_usable_spes (int *cpu_node*)

Definition at line 62 of file info.c.

References `_base_spe_count_physical_spes()`.

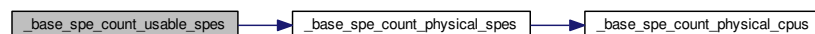
Referenced by `_base_spe_cpu_info_get()`.

```

63 {
64     return _base_spe_count_physical_spes(cpu_node); // FIXME
65 }

```

Here is the call graph for this function:



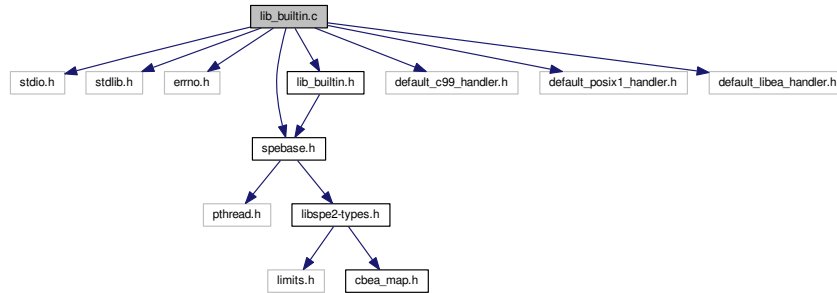
3.14 lib_builtin.c File Reference

```

#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include "spebase.h"
#include "lib_builtin.h"
#include "default_c99_handler.h"
#include "default_posix1_handler.h"
#include "default_libea_handler.h"

```

Include dependency graph for lib_builtin.c:



Macros

- #define [HANDLER_IDX\(x\)](#) ($x \& 0xff$)

Functions

- int [_base_spe_callback_handler_register](#) (void *handler, unsigned int callnum, unsigned int mode)
- int [_base_spe_callback_handler_deregister](#) (unsigned int callnum)
- void * [_base_spe_callback_handler_query](#) (unsigned int callnum)
- int [_base_spe_handle_library_callback](#) (struct [spe_context](#) *spe, int callnum, unsigned int npc)

3.14.1 Macro Definition Documentation

#define [HANDLER_IDX\(x \)](#) ($x \& 0xff$)

Definition at line 29 of file lib_builtin.c.

3.14.2 Function Documentation

int [_base_spe_callback_handler_deregister](#) (unsigned int *callnum*)

unregister a handler function for the specified number NOTE: unregistering a handler from call zero and one is ignored.

Definition at line 78 of file lib_builtin.c.

References [MAX_CALLNUM](#), and [RESERVED](#).

```

79 {
80     errno = 0;
81     if (callnum > MAX\_CALLNUM) {
82         errno = EINVAL;
83         return -1;
84     }
85     if (callnum < RESERVED) {
86         errno = EACCES;
87         return -1;
88     }
89     if (handlers[callnum] == NULL) {
90         errno = ESRCH;
91         return -1;
92     }
93
94     handlers[callnum] = NULL;
95     return 0;
96 }

```

void* _base_spe_callback_handler_query (unsigned int *callnum*)

query a handler function for the specified number

Definition at line 98 of file lib_builtin.c.

References MAX_CALLNUM.

```

99 {
100     errno = 0;
101
102     if (callnum > MAX_CALLNUM) {
103         errno = EINVAL;
104         return NULL;
105     }
106     if (handlers[callnum] == NULL) {
107         errno = ESRCH;
108         return NULL;
109     }
110     return handlers[callnum];
111 }
```

int _base_spe_callback_handler_register (void **handler*, unsigned int *callnum*, unsigned int *mode*)

register a handler function for the specified number NOTE: registering a handler to call zero and one is ignored.

Definition at line 40 of file lib_builtin.c.

References MAX_CALLNUM, RESERVED, SPE_CALLBACK_NEW, and SPE_CALLBACK_UPDATE.

```

41 {
42     errno = 0;
43
44     if (callnum > MAX_CALLNUM) {
45         errno = EINVAL;
46         return -1;
47     }
48
49     switch(mode){
50     case SPE_CALLBACK_NEW:
51         if (callnum < RESERVED) {
52             errno = EACCES;
53             return -1;
54         }
55         if (handlers[callnum] != NULL) {
56             errno = EACCES;
57             return -1;
58         }
59         handlers[callnum] = handler;
60         break;
61
62     case SPE_CALLBACK_UPDATE:
63         if (handlers[callnum] == NULL) {
64             errno = ESRCH;
65             return -1;
66         }
67         handlers[callnum] = handler;
68         break;
69     default:
70         errno = EINVAL;
71         return -1;
72         break;
73     }
74     return 0;
75 }
76 }
```

int _base_spe_handle_library_callback (struct spe_context **spe*, int *callnum*, unsigned int *npc*)

Definition at line 113 of file lib_builtin.c.

References spe_context::base_private, DEBUG_PRINTF, spe_context_base_priv::flags, spe_context_base_priv::mem_mmap_base, SPE_EMULATE_PARAM_BUFFER, and SPE_ISOLATE_EMULATE.

Referenced by _base_spe_context_run().

```

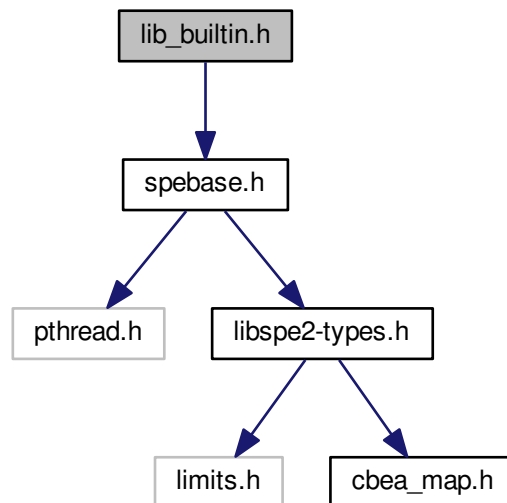
115 {
116     int (*handler)(void *, unsigned int);
117     int rc;
118
119     errno = 0;
120     if (!handlers[callnum]) {
121         DEBUG.PRINTF ("No SPE library handler registered for this call.\n");
122         errno=ENOSYS;
123         return -1;
124     }
125
126     handler=handlers[callnum];
127
128     /* For emulated isolation mode, position the
129      * npc so that the buffer for the PPE-assisted
130      * library calls can be accessed. */
131     if (spe->base.private->flags & SPE.ISOLATE.EMULATE)
132         npc = SPE.EMULATE.PARAM.BUFFER;
133
134     rc = handler(spe->base.private->mem_mmap_base, npc);
135     if (rc) {
136         DEBUG.PRINTF ("SPE library call unsupported.\n");
137         errno=ENOSYS;
138         return rc;
139     }
140     return 0;
141 }

```

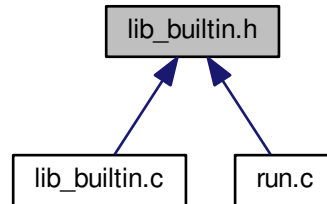
3.15 lib_builtin.h File Reference

```
#include "spebase.h"
```

Include dependency graph for lib_builtin.h:



This graph shows which files directly or indirectly include this file:



Macros

- `#define MAX_CALLNUM 255`
- `#define RESERVED 4`

Functions

- `int _base_spe_handle_library_callback (struct spe_context *spe, int callnum, unsigned int npc)`

3.15.1 Macro Definition Documentation

`#define MAX_CALLNUM 255`

Definition at line 25 of file lib_builtin.h.

Referenced by `_base_spe_callback_handler_deregister()`, `_base_spe_callback_handler_query()`, and `_base_spe_callback_handler_register()`.

`#define RESERVED 4`

Definition at line 26 of file lib_builtin.h.

Referenced by `_base_spe_callback_handler_deregister()`, and `_base_spe_callback_handler_register()`.

3.15.2 Function Documentation

`int _base_spe_handle_library_callback (struct spe_context *spe, int callnum, unsigned int npc)`

Definition at line 113 of file lib_builtin.c.

References `spe_context::base_private`, `DEBUG_PRINTF`, `spe_context_base_priv::flags`, `spe_context_base_priv::mem_mmap_base`, `SPE_EMULATE_PARAM_BUFFER`, and `SPE_ISOLATE_EMULATE`.

Referenced by `_base_spe_context_run()`.

```

115 {
116     int (*handler)(void *, unsigned int);
117     int rc;
118
119     errno = 0;
120     if (!handlers[callnum]) {
121         DEBUG_PRINTF ("No SPE library handler registered for this call.\n");
122         errno=ENOSYS;
123         return -1;
124     }
125

```

```

126     handler=handlers[callnum];
127
128     /* For emulated isolation mode, position the
129     * npc so that the buffer for the PPE-assisted
130     * library calls can be accessed. */
131     if (spe->base.private->flags & SPE_ISOLATE_EMULATE)
132         npc = SPE_EMULATE_PARAM_BUFFER;
133
134     rc = handler(spe->base.private->mem_mmap_base, npc);
135     if (rc) {
136         DEBUG_PRINTF ("SPE library call unsupported.\n");
137         errno=ENOSYS;
138         return rc;
139     }
140     return 0;
141 }

```

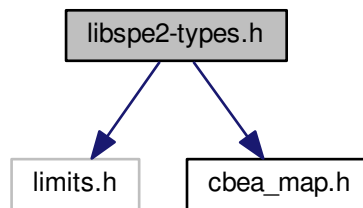
3.16 libspe2-types.h File Reference

```

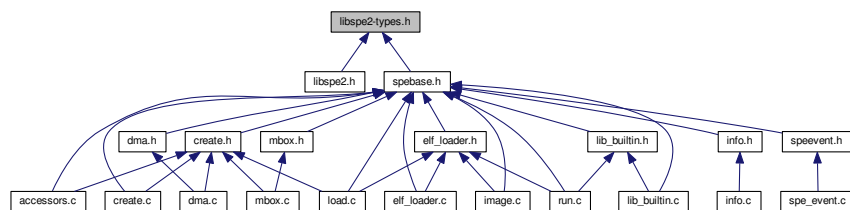
#include <limits.h>
#include "cbea_map.h"

```

Include dependency graph for libspe2-types.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct [spe_program_handle](#)
- struct [spe_context](#)
- struct [spe_gang_context](#)
- struct [spe_stop_info](#)
- union [spe_event_data](#)

- struct `spe_event_unit`

Macros

- #define `SPE_CFG_SIGNOTIFY1_OR` 0x00000010
- #define `SPE_CFG_SIGNOTIFY2_OR` 0x00000020
- #define `SPE_MAP_PS` 0x00000040
- #define `SPE_ISOLATE` 0x00000080
- #define `SPE_ISOLATE_EMULATE` 0x00000100
- #define `SPE_EVENTS_ENABLE` 0x00001000
- #define `SPE_AFFINITY_MEMORY` 0x00002000
- #define `SPE_EXIT` 1
- #define `SPE_STOP_AND_SIGNAL` 2
- #define `SPE_RUNTIME_ERROR` 3
- #define `SPE_RUNTIME_EXCEPTION` 4
- #define `SPE_RUNTIME_FATAL` 5
- #define `SPE_CALLBACK_ERROR` 6
- #define `SPE_ISOLATION_ERROR` 7
- #define `SPE_SPU_STOPPED_BY_STOP` 0x02 /* INTERNAL USE ONLY */
- #define `SPE_SPU_HALT` 0x04
- #define `SPE_SPU_WAITING_ON_CHANNEL` 0x08 /* INTERNAL USE ONLY */
- #define `SPE_SPU_SINGLE_STEP` 0x10
- #define `SPE_SPU_INVALID_INSTR` 0x20
- #define `SPE_SPU_INVALID_CHANNEL` 0x40
- #define `SPE_DMA_ALIGNMENT` 0x0008
- #define `SPE_DMA_SEGMENTATION` 0x0020
- #define `SPE_DMA_STORAGE` 0x0040
- #define `SPE_INVALID_DMA` 0x0800
- #define `SIGSPE` SIGURG
- #define `SPE_EVENT_OUT_INTR_MBOX` 0x00000001
- #define `SPE_EVENT_IN_MBOX` 0x00000002
- #define `SPE_EVENT_TAG_GROUP` 0x00000004
- #define `SPE_EVENT_SPE_STOPPED` 0x00000008
- #define `SPE_EVENT_ALL_EVENTS`
- #define `SPE_MBOX_ALL_BLOCKING` 1
- #define `SPE_MBOX_ANY_BLOCKING` 2
- #define `SPE_MBOX_ANY_NONBLOCKING` 3
- #define `SPE_TAG_ALL` 1
- #define `SPE_TAG_ANY` 2
- #define `SPE_TAG_IMMEDIATE` 3
- #define `SPE_DEFAULT_ENTRY` UINT_MAX
- #define `SPE_RUN_USER_REGS` 0x00000001 /* 128b user data for r3-5. */
- #define `SPE_NO_CALLBACKS` 0x00000002
- #define `SPE_CALLBACK_NEW` 1
- #define `SPE_CALLBACK_UPDATE` 2
- #define `SPE_COUNT_PHYSICAL_CPU_NODES` 1
- #define `SPE_COUNT_PHYSICAL_SPES` 2
- #define `SPE_COUNT_USABLE_SPES` 3
- #define `SPE_SIG_NOTIFY_REG_1` 0x0001
- #define `SPE_SIG_NOTIFY_REG_2` 0x0002

Typedefs

- typedef struct [spe_program_handle](#) [spe_program_handle_t](#)
- typedef struct [spe_context](#) * [spe_context_ptr_t](#)
- typedef struct [spe_gang_context](#) * [spe_gang_context_ptr_t](#)
- typedef struct [spe_stop_info](#) [spe_stop_info_t](#)
- typedef union [spe_event_data](#) [spe_event_data_t](#)
- typedef struct [spe_event_unit](#) [spe_event_unit_t](#)
- typedef void * [spe_event_handler_ptr_t](#)
- typedef int [spe_event_handler_t](#)

Enumerations

- enum [ps_area](#) {
[SPE_MSSYNC_AREA](#), [SPE_MFC_COMMAND_AREA](#), [SPE_CONTROL_AREA](#), [SPE_SIG_NOTIFY_1_AREA](#),
[SPE_SIG_NOTIFY_2_AREA](#) }

3.16.1 Macro Definition Documentation

#define SIGSPE SIGURG

SIGSPE maps to SIGURG

Definition at line 219 of file [libspe2-types.h](#).

#define SPE_AFFINITY_MEMORY 0x00002000

Definition at line 182 of file [libspe2-types.h](#).

Referenced by [_base_spe_context_create\(\)](#).

#define SPE_CALLBACK_ERROR 6

Definition at line 194 of file [libspe2-types.h](#).

Referenced by [_base_spe_context_run\(\)](#).

#define SPE_CALLBACK_NEW 1

Definition at line 260 of file [libspe2-types.h](#).

Referenced by [_base_spe_callback_handler_register\(\)](#).

#define SPE_CALLBACK_UPDATE 2

Definition at line 261 of file [libspe2-types.h](#).

Referenced by [_base_spe_callback_handler_register\(\)](#).

#define SPE_CFG_SIGNOTIFY1_OR 0x00000010

Flags for [spe_context_create](#)

Definition at line 176 of file [libspe2-types.h](#).

Referenced by [_base_spe_context_create\(\)](#).

#define SPE_CFG_SIGNOTIFY2_OR 0x00000020

Definition at line 177 of file [libspe2-types.h](#).

Referenced by [_base_spe_context_create\(\)](#).

Definition at line 265 of file libspe2-types.h.
Referenced by `_base_spe_cpu_info_get()`.

Definition at line 266 of file libspe2-types.h.
Referenced by `_base_spe_cpu_info_get()`.

Definition at line 267 of file libspe2-types.h.
Referenced by `_base_spe_cpu_info_get()`.

Flags for `_base_spe_context_run`
 Definition at line 253 of file `libspe2-types.h`.
 Referenced by `_base_spe_context_run()`.

Runtime exceptions
Definition at line 210 of file libspe2-types.h.

Definition at line 212 of file libspe2-types.h.

Definition at line 213 of file libspe2-types.h.

```
SPE_EVENT_OUT_INTR_MBOX | \
    \
    \
    \
    SPE_EVENT_IN_MBOX |
    SPE_EVENT_TAG_GROUP |
    SPE_EVENT_SPE_STOPPED
```

Definition at line 229 of file libspe2-types.h.

Definition at line 225 of file libspe2-types.h.
Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

Supported SPE events
 Definition at line 224 of file libspe2-types.h.
 Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

#define SPE_EVENT_SPE_STOPPED 0x00000008

Definition at line 227 of file libspe2-types.h.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

#define SPE_EVENT_TAG_GROUP 0x00000004

Definition at line 226 of file libspe2-types.h.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

#define SPE_EVENTS_ENABLE 0x00001000

Definition at line 181 of file libspe2-types.h.

Referenced by `_base_spe_context_create()`, and `_base_spe_context_run()`.

#define SPE_EXIT 1

Symbolic constants for stop reasons as returned in `spe_stop_info_t`

Definition at line 189 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_INVALID_DMA 0x0800

Definition at line 214 of file libspe2-types.h.

#define SPE_ISOLATE 0x00000080

Definition at line 179 of file libspe2-types.h.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, and `_base_spe_program_load()`.

#define SPE_ISOLATE_EMULATE 0x00000100

Definition at line 180 of file libspe2-types.h.

Referenced by `_base_spe_context_create()`, `_base_spe_context_run()`, `_base_spe_handle_library_callback()`, and `_base_spe_program_load()`.

#define SPE_ISOLATION_ERROR 7

Definition at line 195 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_MAP_PS 0x00000040

Definition at line 178 of file libspe2-types.h.

Referenced by `_base_spe_context_create()`, `_base_spe_in_mbox_status()`, `_base_spe_in_mbox_write()`, `_base_spe_mfcio_tag_status_read()`, `_base_spe_mssync_start()`, `_base_spe_mssync_status()`, `_base_spe_out_intr_mbox_status()`, `_base_spe_out_mbox_read()`, `_base_spe_out_mbox_status()`, `_base_spe_signal_write()`, and `_event_spe_event_handler_register()`.

#define SPE_MBOX_ALL_BLOCKING 1

Behavior flags for mailbox read/write functions

Definition at line 237 of file libspe2-types.h.

Referenced by `_base_spe_in_mbox_write()`, and `_base_spe_out_intr_mbox_read()`.

#define SPE_MBOX_ANY_BLOCKING 2

Definition at line 238 of file libspe2-types.h.

Referenced by `_base_spe_in_mbox_write()`, and `_base_spe_out_intr_mbox_read()`.

#define SPE_MBOX_ANY_NONBLOCKING 3

Definition at line 239 of file libspe2-types.h.

Referenced by `_base_spe_in_mbox_write()`, and `_base_spe_out_intr_mbox_read()`.

#define SPE_NO_CALLBACKS 0x00000002

Definition at line 255 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_RUN_USER_REGS 0x00000001 /* 128b user data for r3-5. */

Definition at line 254 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_RUNTIME_ERROR 3

Definition at line 191 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_RUNTIME_EXCEPTION 4

Definition at line 192 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_RUNTIME_FATAL 5

Definition at line 193 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_SIG_NOTIFY_REG_1 0x0001

Signal Targets

Definition at line 272 of file libspe2-types.h.

Referenced by `_base_spe_signal_write()`.

#define SPE_SIG_NOTIFY_REG_2 0x0002

Definition at line 273 of file libspe2-types.h.

Referenced by `_base_spe_signal_write()`.

#define SPE_SPU_HALT 0x04

Definition at line 201 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_SPU_INVALID_CHANNEL 0x40

Definition at line 205 of file libspe2-types.h.

Referenced by `_base_spe_context_run()`.

#define SPE_SPU_INVALID_INSTR 0x20

Definition at line 204 of file libspe2-types.h.
 Referenced by `_base_spe_context_run()`.

#define SPE_SPU_SINGLE_STEP 0x10

Definition at line 203 of file libspe2-types.h.

#define SPE_SPU_STOPPED_BY_STOP 0x02 /* INTERNAL USE ONLY */

Runtime errors

Definition at line 200 of file libspe2-types.h.
 Referenced by `_base_spe_context_run()`.

#define SPE_SPU_WAITING_ON_CHANNEL 0x08 /* INTERNAL USE ONLY */

Definition at line 202 of file libspe2-types.h.
 Referenced by `_base_spe_context_run()`.

#define SPE_STOP_AND_SIGNAL 2

Definition at line 190 of file libspe2-types.h.
 Referenced by `_base_spe_context_run()`.

#define SPE_TAG_ALL 1

Behavior flags tag status functions

Definition at line 245 of file libspe2-types.h.
 Referenced by `_base_spe_mfcio_tag_status_read()`.

#define SPE_TAG_ANY 2

Definition at line 246 of file libspe2-types.h.
 Referenced by `_base_spe_mfcio_tag_status_read()`.

#define SPE_TAG_IMMEDIATE 3

Definition at line 247 of file libspe2-types.h.
 Referenced by `_base_spe_mfcio_tag_status_read()`.

3.16.2 Typedef Documentation**typedef struct spe_context* spe_context_ptr_t**

`spe_context_ptr_t` This pointer serves as the identifier for a specific SPE context throughout the API (where needed)

Definition at line 83 of file libspe2-types.h.

typedef union spe_event_data spe_event_data_t

`spe_event_data_t` User data to be associated with an event

typedef void* spe_event_handler_ptr_t

Definition at line 159 of file libspe2-types.h.

typedef int spe_event_handler_t

Definition at line 160 of file libspe2-types.h.

typedef struct spe_event_unit spe_event_unit_t

spe_event_t

typedef struct spe_gang_context* spe_gang_context_ptr_t

spe_gang_context_ptr_t This pointer serves as the identifier for a specific SPE gang context throughout the API (where needed)

Definition at line 106 of file libspe2-types.h.

typedef struct spe_program_handle spe_program_handle_t

SPE program handle Structure [spe_program_handle](#) per CESOF specification libspe2 applications usually only keep a pointer to the program handle and do not use the structure directly.

typedef struct spe_stop_info spe_stop_info_t

spe_stop_info_t

3.16.3 Enumeration Type Documentation

enum ps_area

Enumerator

SPE_MSSYNC_AREA

SPE_MFC_COMMAND_AREA

SPE_CONTROL_AREA

SPE_SIG_NOTIFY_1_AREA

SPE_SIG_NOTIFY_2_AREA

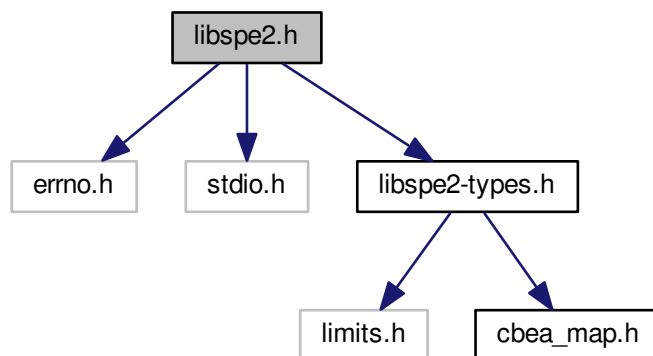
Definition at line 171 of file libspe2-types.h.

```
171 { SPE_MSSYNC_AREA, SPE_MFC_COMMAND_AREA,
    SPE_CONTROL_AREA, SPE_SIG_NOTIFY_1_AREA,
    SPE_SIG_NOTIFY_2_AREA };
```

3.17 libspe2.h File Reference

```
#include <errno.h>
#include <stdio.h>
#include "libspe2-types.h"
```

Include dependency graph for `libspe2.h`:



Functions

- `spe_context_ptr_t spe_context_create` (unsigned int flags, `spe_gang_context_ptr_t` gang)
- `spe_context_ptr_t spe_context_create_affinity` (unsigned int flags, `spe_context_ptr_t` affinity_neighbor, `spe_gang_context_ptr_t` gang)
- `int spe_context_destroy` (`spe_context_ptr_t` spe)
- `spe_gang_context_ptr_t spe_gang_context_create` (unsigned int flags)
- `int spe_gang_context_destroy` (`spe_gang_context_ptr_t` gang)
- `spe_program_handle_t * spe_image_open` (const char *filename)
- `int spe_image_close` (`spe_program_handle_t` *program)
- `int spe_program_load` (`spe_context_ptr_t` spe, `spe_program_handle_t` *program)
- `int spe_context_run` (`spe_context_ptr_t` spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, `spe_stop_info_t` *stopinfo)
- `int spe_stop_info_read` (`spe_context_ptr_t` spe, `spe_stop_info_t` *stopinfo)
- `spe_event_handler_ptr_t spe_event_handler_create` (void)
- `int spe_event_handler_destroy` (`spe_event_handler_ptr_t` evhandler)
- `int spe_event_handler_register` (`spe_event_handler_ptr_t` evhandler, `spe_event_unit_t` *event)
- `int spe_event_handler_deregister` (`spe_event_handler_ptr_t` evhandler, `spe_event_unit_t` *event)
- `int spe_event_wait` (`spe_event_handler_ptr_t` evhandler, `spe_event_unit_t` *events, int max_events, int timeout)
- `int spe_mfcio_put` (`spe_context_ptr_t` spe, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int spe_mfcio_putb` (`spe_context_ptr_t` spe, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int spe_mfcio_putf` (`spe_context_ptr_t` spe, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int spe_mfcio_get` (`spe_context_ptr_t` spe, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int spe_mfcio_getb` (`spe_context_ptr_t` spe, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

- int `spe_mfcio_getf` (`spe_context_ptr_t` spe, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- int `spe_mfcio_tag_status_read` (`spe_context_ptr_t` spe, unsigned int mask, unsigned int behavior, unsigned int *tag_status)
- int `spe_out_mbox_read` (`spe_context_ptr_t` spe, unsigned int *mbox_data, int count)
- int `spe_out_mbox_status` (`spe_context_ptr_t` spe)
- int `spe_in_mbox_write` (`spe_context_ptr_t` spe, unsigned int *mbox_data, int count, unsigned int behavior)
- int `spe_in_mbox_status` (`spe_context_ptr_t` spe)
- int `spe_out_intr_mbox_read` (`spe_context_ptr_t` spe, unsigned int *mbox_data, int count, unsigned int behavior)
- int `spe_out_intr_mbox_status` (`spe_context_ptr_t` spe)
- int `spe_mssync_start` (`spe_context_ptr_t` spe)
- int `spe_mssync_status` (`spe_context_ptr_t` spe)
- int `spe_signal_write` (`spe_context_ptr_t` spe, unsigned int signal_reg, unsigned int data)
- void * `spe_ls_area_get` (`spe_context_ptr_t` spe)
- int `spe_ls_size_get` (`spe_context_ptr_t` spe)
- void * `spe_ps_area_get` (`spe_context_ptr_t` spe, enum `ps_area` area)
- int `spe_callback_handler_register` (void *handler, unsigned int callnum, unsigned int mode)
- int `spe_callback_handler_deregister` (unsigned int callnum)
- void * `spe_callback_handler_query` (unsigned int callnum)
- int `spe_cpu_info_get` (int info_requested, int cpu_node)

3.17.1 Function Documentation

```

int spe_callback_handler_deregister ( unsigned int callnum )

void* spe_callback_handler_query ( unsigned int callnum )

int spe_callback_handler_register ( void * handler, unsigned int callnum, unsigned int mode )

spe_context_ptr_t spe_context_create ( unsigned int flags, spe_gang_context_ptr_t gang )

spe_context_ptr_t spe_context_create_affinity ( unsigned int flags, spe_context_ptr_t affinity_neighbor,
spe_gang_context_ptr_t gang )

int spe_context_destroy ( spe_context_ptr_t spe )

int spe_context_run ( spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp,
void * envp, spe_stop_info_t * stopinfo )

int spe_cpu_info_get ( int info_requested, int cpu_node )

spe_event_handler_ptr_t spe_event_handler_create ( void )

int spe_event_handler_deregister ( spe_event_handler_ptr_t evhandler, spe_event_unit_t * event )

int spe_event_handler_destroy ( spe_event_handler_ptr_t evhandler )

int spe_event_handler_register ( spe_event_handler_ptr_t evhandler, spe_event_unit_t * event )

int spe_event_wait ( spe_event_handler_ptr_t evhandler, spe_event_unit_t * events, int max_events,
int timeout )

spe_gang_context_ptr_t spe_gang_context_create ( unsigned int flags )

int spe_gang_context_destroy ( spe_gang_context_ptr_t gang )

int spe_image_close ( spe_program_handle_t * program )

spe_program_handle_t* spe_image_open ( const char * filename )

int spe_in_mbox_status ( spe_context_ptr_t spe )

int spe_in_mbox_write ( spe_context_ptr_t spe, unsigned int * mbox_data, int count, unsigned int
behavior )

void* spe_ls_area_get ( spe_context_ptr_t spe )

int spe_ls_size_get ( spe_context_ptr_t spe )

int spe_mfcio_get ( spe_context_ptr_t spe, unsigned int ls, void * ea, unsigned int size, unsigned int
tag, unsigned int tid, unsigned int rid )

int spe_mfcio_getb ( spe_context_ptr_t spe, unsigned int ls, void * ea, unsigned int size, unsigned
int tag, unsigned int tid, unsigned int rid )

int spe_mfcio_getf ( spe_context_ptr_t spe, unsigned int ls, void * ea, unsigned int size, unsigned int
tag, unsigned int tid, unsigned int rid )

int spe_mfcio_put ( spe_context_ptr_t spe, unsigned int ls, void * ea, unsigned int size, unsigned int
tag, unsigned int tid, unsigned int rid )

int spe_mfcio_putb ( spe_context_ptr_t spe, unsigned int ls, void * ea, unsigned int size, unsigned
int tag, unsigned int tid, unsigned int rid )

int spe_mfcio_putf ( spe_context_ptr_t spe, unsigned int ls, void * ea, unsigned int size, unsigned
int tag, unsigned int tid, unsigned int rid )

int spe_mfcio_tag_status_read ( spe_context_ptr_t spe, unsigned int mask, unsigned int behavior,
unsigned int * tag_status )

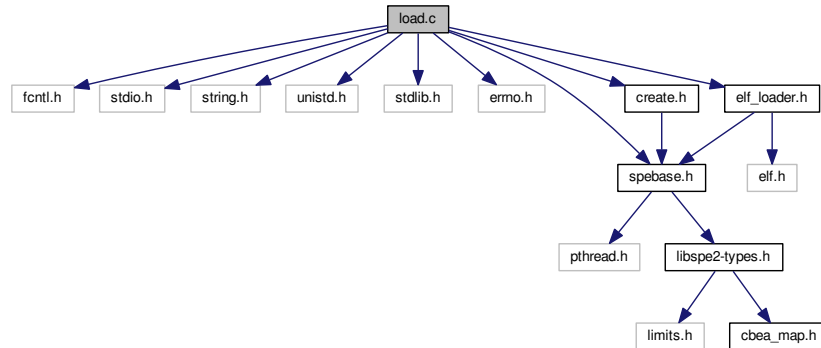
int spe_mssync.start ( spe_context_ptr_t spe )

int spe_mssync.status ( spe_context_ptr_t spe )

int spe_out_intr_mbox_read ( spe_context_ptr_t spe, unsigned int * mbox_data, int count, unsigned
int behavior )

```

Include dependency graph for load.c:



Macros

- `#define SPE_EMULATED_LOADER_FILE "/usr/lib/spe/emulated-loader.bin"`

Functions

- `void _base_spe_program_load_complete (spe_context_ptr_t spectx)`
- `int _base_spe_emulated_loader_present (void)`
- `int _base_spe_program_load (spe_context_ptr_t spe, spe_program_handle_t *program)`

3.18.1 Macro Definition Documentation

`#define SPE_EMULATED_LOADER_FILE "/usr/lib/spe/emulated-loader.bin"`

Definition at line 31 of file load.c.

3.18.2 Function Documentation

`int _base_spe_emulated_loader_present (void)`

Check if the emulated loader is present in the filesystem

Returns

Non-zero if the loader is available, otherwise zero.

Definition at line 159 of file load.c.

References `_base_spe_verify_spe_elf_image()`.

Referenced by `_base_spe_context_create()`.

```

160 {
161     spe_program_handle_t *loader = emulated_loader_program();
162
163     if (!loader)
164         return 0;
165
166     return !_base_spe_verify_spe_elf_image(loader);
167 }
  
```

Here is the call graph for this function:



int _base_spe_program_load (spe_context_ptr_t *spectx*, spe_program_handle_t * *program*)

_base_spe_program_load loads an ELF image into a context

Parameters

<i>spectx</i>	Specifies the SPE context
<i>program</i>	handle to the ELF image

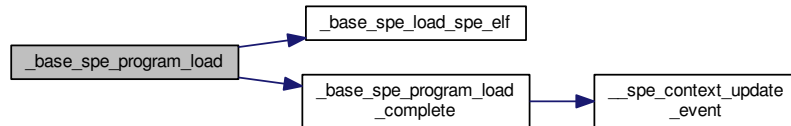
Definition at line 203 of file load.c.

References _base_spe.load_spe_elf(), _base_spe.program_load_complete(), spe_context::base_private, DEBUG_PRINTF, spe_context_base_priv::emulated_entry, spe_ld_info::entry, spe_context_base_priv::entry, spe_context_base_priv::flags, spe_context_base_priv::loaded_program, spe_context_base_priv::mem_mmap_base, SPE_ISOLATE, and SPE_ISOLATE_EMULATE.

```

204 {
205     int rc = 0;
206     struct spe_ld_info ld_info;
207
208     spe->base_private->loaded_program = program;
209
210     if (spe->base_private->flags & SPE_ISOLATE) {
211         rc = spe.start.isolated.app(spe, program);
212     } else if (spe->base_private->flags &
213 SPE_ISOLATE_EMULATE) {
214         rc = spe.start.emulated.isolated.app(spe, program, &ld_info);
215     } else {
216         rc = _base_spe.load_spe_elf(program,
217                                     spe->base_private->mem_mmap_base, &ld_info);
218         if (!rc)
219             _base_spe.program_load_complete(spe);
220     }
221
222     if (rc != 0) {
223         DEBUG_PRINTF ("Load SPE ELF failed..\n");
224         return -1;
225     }
226
227     spe->base_private->entry = ld_info.entry;
228     spe->base_private->emulated_entry = ld_info.entry;
229
230     return 0;
231 }
232
  
```

Here is the call graph for this function:



void _base_spe_program_load_complete (spe_context_ptr_t spectx)

Register the SPE program's start address with the oprofile and gdb, by writing to the object-id file.

Definition at line 38 of file load.c.

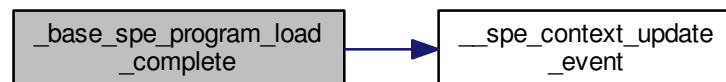
References __spe_context_update_event(), spe_context::base_private, DEBUG_PRINTF, spe_program_handle::elf_image, spe_context_base_priv::fd_spe_dir, and spe_context_base_priv::loaded_program.

Referenced by _base_spe_context_run(), and _base_spe_program_load().

```

39 {
40     int objfd, len;
41     char buf[20];
42     spe_program_handle_t *program;
43
44     program = spectx->base_private->loaded_program;
45
46     if (!program || !program->elf_image) {
47         DEBUG_PRINTF("%s called, but no program loaded\n", __func__);
48         return;
49     }
50
51     objfd = openat(spectx->base_private->fd_spe_dir, "object-id", O_RDWR);
52     if (objfd < 0)
53         return;
54
55     len = sprintf(buf, "%p", program->elf_image);
56     write(objfd, buf, len + 1);
57     close(objfd);
58
59     __spe_context_update_event();
60 }
  
```

Here is the call graph for this function:

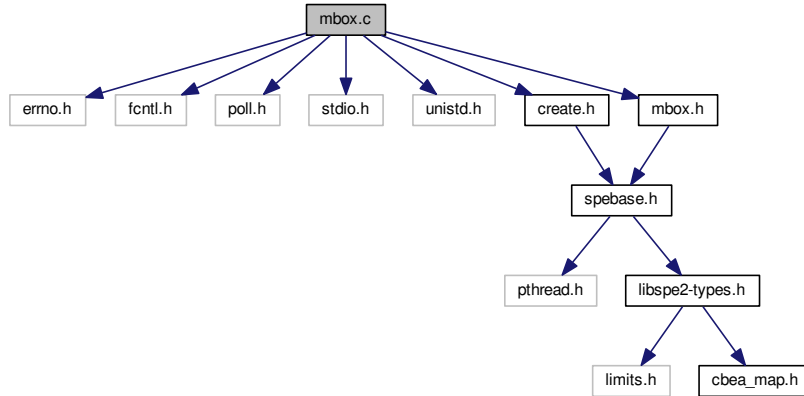


3.19 mbox.c File Reference

```
#include <errno.h>
```

```
#include <fcntl.h>
#include <poll.h>
#include <stdio.h>
#include <unistd.h>
#include "create.h"
#include "mbox.h"
```

Include dependency graph for mbox.c:



Functions

- `int _base_spe_out_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[], int count)`
- `int _base_spe_in_mbox_write (spe_context_ptr_t spectx, unsigned int *mbox_data, int count, int behavior_flag)`
- `int _base_spe_in_mbox_status (spe_context_ptr_t spectx)`
- `int _base_spe_out_mbox_status (spe_context_ptr_t spectx)`
- `int _base_spe_out_intr_mbox_status (spe_context_ptr_t spectx)`
- `int _base_spe_out_intr_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[], int count, int behavior_flag)`
- `int _base_spe_signal_write (spe_context_ptr_t spectx, unsigned int signal_reg, unsigned int data)`

3.19.1 Function Documentation

int _base_spe_in_mbox_status (spe_context_ptr_t spectx)

The `_base_spe_in_mbox_status` function fetches the status of the SPU inbound mailbox for the SPE thread specified by the `speid` parameter. A 0 value is return if the mailbox is full. A non-zero value specifies the number of available (32-bit) mailbox entries.

Parameters

<i>spectx</i>	Specifies the SPE context whose mailbox status is to be read.
---------------	---

Returns

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

See Also

`spe_read_out_mbox`, `spe_write_in_mbox`, `read` (2)

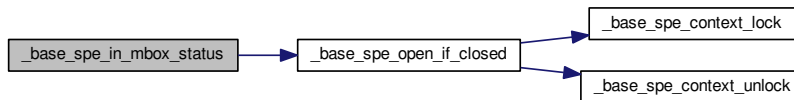
Definition at line 202 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `FD_WBOX_STAT`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, and `spe_spu_control_area::SPU_Mbox_Stat`.

```

203 {
204     int rc, ret;
205     volatile struct spe_spu_control_area *cntl_area =
206         spectx->base_private->cntl_mmap_base;
207
208     if (spectx->base_private->flags & SPE_MAP_PS) {
209         ret = (cntl_area->SPU_Mbox_Stat >> 8) & 0xFF;
210     } else {
211         rc = read(_base_spe_open_if_closed(spectx,
212     FD_WBOX_STAT, 0), &ret, 4);
213         if (rc != 4)
214             ret = -1;
215     }
216     return ret;
217 }
218 }
```

Here is the call graph for this function:



`int _base_spe_in_mbox_write (spe_context_ptr_t spectx, unsigned int * mbox_data, int count, int behavior_flag)`

Definition at line 112 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_WBOX`, `FD_WBOX_NB`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, `SPE_MBOX_ALL_BLOCKING`, `SPE_MBOX_ANY_BLOCKING`, and `SPE_MBOX_ANY_NONBLOCKING`.

```

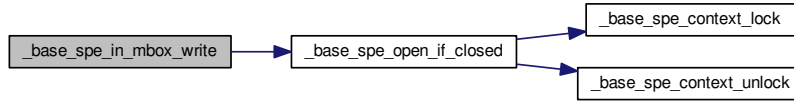
116 {
117     int rc;
118     int total;
119     unsigned int *aux;
120     struct pollfd fds;
121
122     if (mbox_data == NULL || count < 1){
123         errno = EINVAL;
124         return -1;
125     }
126
127     switch (behavior_flag) {
128     case SPE_MBOX_ALL_BLOCKING: // write all, even if blocking
129         total = rc = 0;
130         if (spectx->base_private->flags & SPE_MAP_PS) {
131             do {
132                 aux = mbox_data + total;
133                 total += _base_spe_in_mbox_write_ps(spectx, aux, count - total);
134                 if (total < count) { // we could not write everything, wait for space
135                     fds.fd = _base_spe_open_if_closed(spectx,
136     FD_WBOX, 0);
```

```

136         fds.events = POLLOUT;
137         rc = poll(&fds, 1, -1);
138         if (rc == -1 )
139             return -1;
140     }
141     } while (total < count);
142 } else {
143     while (total < 4*count) {
144         rc = write(_base_spe_open_if_closed(spectx,
145         FD.WBOX, 0),
146                 (const char *)mbox.data + total, 4*count - total);
147         if (rc == -1) {
148             break;
149         }
150         total += rc;
151     }
152     total /=4;
153     break;
154 }
155 case SPE_MBOX_ANY_BLOCKING: // write at least one, even if blocking
156     total = rc = 0;
157     if (spectx->base_private->flags & SPE_MAP_PS) {
158         do {
159             total = _base_spe_in_mbox.write_ps(spectx, mbox.data, count);
160             if (total == 0) { // we could not anything, wait for space
161                 fds.fd = _base_spe_open_if_closed(spectx,
162                 FD.WBOX, 0);
163                 fds.events = POLLOUT;
164                 rc = poll(&fds, 1, -1);
165                 if (rc == -1 )
166                     return -1;
167             }
168         } while (total == 0);
169     } else {
170         rc = write(_base_spe_open_if_closed(spectx,
171         FD.WBOX, 0), mbox.data, 4*count);
172         total = rc/4;
173     }
174     break;
175 }
176 case SPE_MBOX_ANY_NONBLOCKING: // only write, if non blocking
177     total = rc = 0;
178     // write directly if we map the PS else write via spufs
179     if (spectx->base_private->flags & SPE_MAP_PS) {
180         total = _base_spe_in_mbox.write_ps(spectx, mbox.data, count);
181     } else {
182         rc = write(_base_spe_open_if_closed(spectx,
183         FD.WBOX_NB, 0), mbox.data, 4*count);
184         if (rc == -1 && errno == EAGAIN) {
185             rc = 0;
186             errno = 0;
187         }
188         total = rc/4;
189     }
190     break;
191 }
192 default:
193     errno = EINVAL;
194     return -1;
195 }
196 if (rc == -1) {
197     errno = EIO;
198     return -1;
199 }
200 return total;

```

Here is the call graph for this function:



int _base_spe_out_intr_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[], int count, int behavior_flag)

The `_base_spe_out_intr_mbox_read` function reads the contents of the SPE outbound interrupting mailbox for the SPE context.

Definition at line 255 of file `mbox.c`.

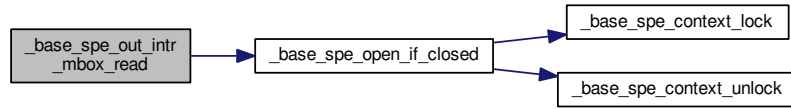
References `_base_spe_open_if_closed()`, `FD_IBOX`, `FD_IBOX_NB`, `SPE_MBOX_ALL_BLOCKING`, `SPE_MBOX_ANY_BLOCKING`, and `SPE_MBOX_ANY_NONBLOCKING`.

```

259 {
260     int rc;
261     int total;
262
263     if (mbox_data == NULL || count < 1){
264         errno = EINVAL;
265         return -1;
266     }
267
268     switch (behavior_flag) {
269     case SPE_MBOX_ALL_BLOCKING: // read all, even if blocking
270         total = rc = 0;
271         while (total < 4*count) {
272             rc = read(_base_spe_open_if_closed(spectx,
273             FD_IBOX, 0),
274                     (char *)mbox_data + total, 4*count - total);
275             if (rc == -1) {
276                 break;
277             }
278             total += rc;
279             break;
280
281     case SPE_MBOX_ANY_BLOCKING: // read at least one, even if blocking
282         total = rc = read(_base_spe_open_if_closed(spectx,
283             FD_IBOX, 0), mbox_data, 4*count);
284         break;
285
286     case SPE_MBOX_ANY_NONBLOCKING: // only read, if non blocking
287         rc = read(_base_spe_open_if_closed(spectx,
288             FD_IBOX_NB, 0), mbox_data, 4*count);
289         if (rc == -1 && errno == EAGAIN) {
290             rc = 0;
291             errno = 0;
292         }
293         total = rc;
294         break;
295
296     default:
297         errno = EINVAL;
298         return -1;
299     }
300
301     if (rc == -1) {
302         errno = EIO;
303         return -1;
304     }
305
306     return rc / 4;
307 }

```


Here is the call graph for this function:



int _base_spe_out_intr_mbox_status (spe_context_ptr_t spectx)

The `_base_spe_out_intr_mbox_status` function fetches the status of the SPU outbound interrupt mailbox for the SPE thread specified by the `speid` parameter. A 0 value is return if the mailbox is empty. A non-zero value specifies the number of 32-bit unread mailbox entries.

Parameters

<i>spectx</i>	Specifies the SPE context whose mailbox status is to be read.
---------------	---

Returns

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

See Also

`spe_read_out_mbox`, `spe_write_in_mbox`, `read (2)`

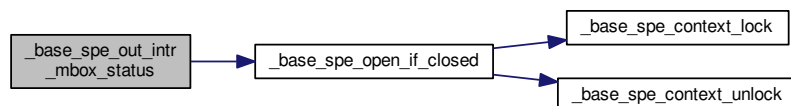
Definition at line 238 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context::base_priv::cntl_mmap_base`, `FD_IBOX_STAT`, `spe_context::base_priv::flags`, `SPE_MAP_PS`, and `spe_spu_control_area::SPU_Mbox_Stat`.

```

239 {
240     int rc, ret;
241     volatile struct spe_spu_control_area *cntl_area =
242         spectx->base_private->cntl_mmap_base;
243
244     if (spectx->base_private->flags & SPE_MAP_PS) {
245         ret = (cntl_area->SPU_Mbox_Stat >> 16) & 0xFF;
246     } else {
247         rc = read(_base_spe_open_if_closed(spectx,
248         FD_IBOX_STAT, 0), &ret, 4);
249         if (rc != 4)
250             ret = -1;
251     }
252     return ret;
253 }
  
```

Here is the call graph for this function:



int _base_spe_out_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[], int count)

The _base_spe_out_mbox_read function reads the contents of the SPE outbound interrupting mailbox for the SPE thread speid.

The call will not block until the read request is satisfied, but instead return up to count currently available mailbox entries.

spe_stat_out_intr_mbox can be called to ensure that data is available prior to reading the outbound interrupting mailbox.

Parameters

<i>spectx</i>	Specifies the SPE thread whose outbound mailbox is to be read.
<i>mbox_data</i>	
<i>count</i>	

Return values

>0	the number of 32-bit mailbox messages read
=0	no data available
-1	error condition and errno is set Possible values for errno: EINVAL speid is invalid Exxxx what else do we need??

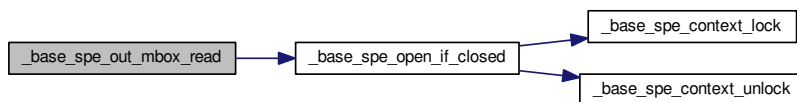
Definition at line 58 of file mbox.c.

References _base_spe_open_if_closed(), spe_context::base_private, DEBUG_PRINTF, FD_MBOX, spe_context_base_priv::flags, and SPE_MAP_PS.

```

61 {
62     int rc;
63
64     if (mbox_data == NULL || count < 1){
65         errno = EINVAL;
66         return -1;
67     }
68
69     if (spectx->base_private->flags & SPE_MAP_PS) {
70         rc = _base_spe_out_mbox_read.ps(spectx, mbox_data, count);
71     } else {
72         rc = read(_base_spe_open_if_closed(spectx,
73 FD_MBOX, 0), mbox_data, count*4);
74         DEBUG_PRINTF("%s read rc: %d\n", __FUNCTION__, rc);
75         if (rc != -1) {
76             rc /= 4;
77         } else {
78             if (errno == EAGAIN ) { // no data ready to be read
79                 errno = 0;
80                 rc = 0;
81             }
82         }
83     }
84     return rc;
85 }
```

Here is the call graph for this function:



int _base_spe_out_mbox_status (spe_context_ptr_t spectx)

The `_base_spe_out_mbox_status` function fetches the status of the SPU outbound mailbox for the SPE thread specified by the `speid` parameter. A 0 value is return if the mailbox is empty. A non-zero value specifies the number of 32-bit unread mailbox entries.

Parameters

<i>spectx</i>	Specifies the SPE context whose mailbox status is to be read.
---------------	---

Returns

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

See Also

`spe_read_out_mbox`, `spe_write_in_mbox`, `read` (2)

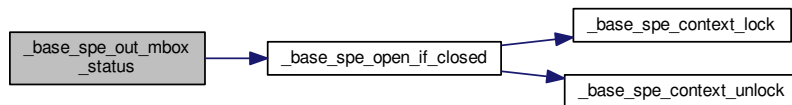
Definition at line 220 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `FD_MBOX_STAT`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, and `spe_spu_control_area::SPU_Mbox_Stat`.

```

221 {
222     int rc, ret;
223     volatile struct spe_spu_control_area *cntl_area =
224         spectx->base_private->cntl_mmap_base;
225
226     if (spectx->base_private->flags & SPE_MAP_PS) {
227         ret = cntl_area->SPU_Mbox_Stat & 0xFF;
228     } else {
229         rc = read(_base_spe_open_if_closed(spectx,
230             FD_MBOX_STAT, 0), &ret, 4);
231         if (rc != 4)
232             ret = -1;
233     }
234     return ret;
235 }
236 }
```

Here is the call graph for this function:

**int _base_spe_signal_write (spe_context_ptr_t spectx, unsigned int signal_reg, unsigned int data)**

The `_base_spe_signal_write` function writes data to the signal notification register specified by `signal_reg` for the SPE thread specified by the `speid` parameter.

Parameters

<i>spectx</i>	Specifies the SPE context whose signal register is to be written to.
<i>signal_reg</i>	Specified the signal notification register to be written. Valid signal notification registers are: SPE.SIG_NOTIFY_REG.1 SPE signal notification register 1 SPE.SIG_NOTIFY_REG.2 SPE signal notification register 2
<i>data</i>	The 32-bit data to be written to the specified signal notification register.

Returns

On success, `spe_write_signal` returns 0. On failure, -1 is returned.

See Also

`spe_get_ps_area`, `spe_write_in_mbox`

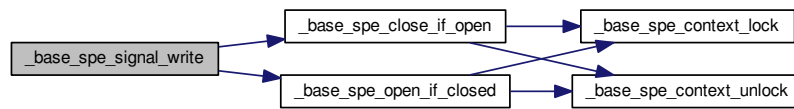
Definition at line 307 of file `mbox.c`.

References `_base_spe_close_if_open()`, `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_SIG1`, `FD_SIG2`, `spe_context_base_priv::flags`, `spe_context_base_priv::signal1_mmap_base`, `spe_context_base_priv::signal2_mmap_base`, `SPE_MAP_PS`, `SPE_SIG_NOTIFY_REG.1`, `SPE_SIG_NOTIFY_REG.2`, `spe_sig_notify_1_area::SPU_Sig_Notify_1`, and `spe_sig_notify_2_area::SPU_Sig_Notify_2`.

```

310 {
311     int rc;
312
313     if (spectx->base_private->flags & SPE_MAP_PS) {
314         if (signal_reg == SPE_SIG_NOTIFY_REG.1) {
315             spe_sig_notify_1_area_t *sig = spectx->
base_private->signal1_mmap_base;
316
317             sig->SPU_Sig_Notify_1 = data;
318         } else if (signal_reg == SPE_SIG_NOTIFY_REG.2) {
319             spe_sig_notify_2_area_t *sig = spectx->
base_private->signal2_mmap_base;
320
321             sig->SPU_Sig_Notify_2 = data;
322         } else {
323             errno = EINVAL;
324             return -1;
325         }
326         rc = 0;
327     } else {
328         if (signal_reg == SPE_SIG_NOTIFY_REG.1)
329             rc = write(_base_spe_open_if_closed(spectx,
FD_SIG1, 0), &data, 4);
330         else if (signal_reg == SPE_SIG_NOTIFY_REG.2)
331             rc = write(_base_spe_open_if_closed(spectx,
FD_SIG2, 0), &data, 4);
332         else {
333             errno = EINVAL;
334             return -1;
335         }
336
337         if (rc == 4)
338             rc = 0;
339
340         if (signal_reg == SPE_SIG_NOTIFY_REG.1)
341             _base_spe_close_if_open(spectx,
FD_SIG1);
342         else if (signal_reg == SPE_SIG_NOTIFY_REG.2)
343             _base_spe_close_if_open(spectx,
FD_SIG2);
344     }
345     return rc;
346 }
347 }
```

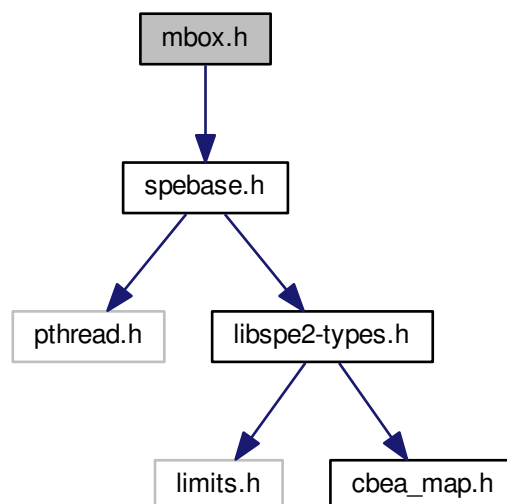
Here is the call graph for this function:



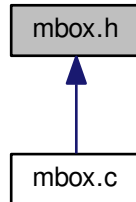
3.20 mbox.h File Reference

```
#include "spebase.h"
```

Include dependency graph for mbox.h:



This graph shows which files directly or indirectly include this file:



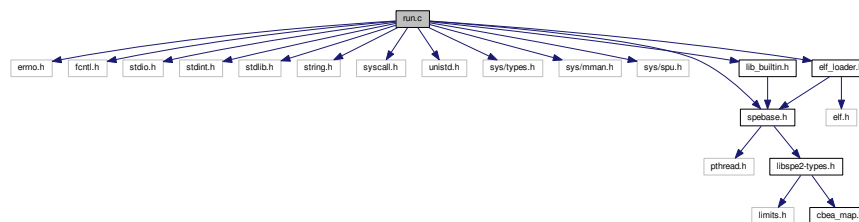
3.21 run.c File Reference

```

#include <errno.h>
#include <fcntl.h>
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <syscall.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/mman.h>
#include <sys/spu.h>
#include "elf_loader.h"
#include "lib_builtin.h"
#include "spebase.h"

```

Include dependency graph for run.c:



Data Structures

- struct [spe_context_info](#)

Macros

- #define [GNU_SOURCE](#) 1

Functions

- `int _base_spe_context_run (spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)`

Variables

- `__thread struct spe_context_info * __spe_current_active_context`

3.21.1 Macro Definition Documentation

`#define GNU_SOURCE 1`

Definition at line 20 of file run.c.

3.21.2 Function Documentation

`int _base_spe_context_run (spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp, void * envp, spe_stop_info_t * stopinfo)`

`_base_spe_context_run` starts execution of an SPE context with a loaded image

Parameters

<i>spectx</i>	Specifies the SPE context
<i>entry</i>	entry point for the SPE program. If set to 0, entry point is determined by the ELF loader.
<i>runflags</i>	valid values are: SPE_RUN_USER_REGS Specifies that the SPE setup registers r3, r4, and r5 are initialized with the 48 bytes pointed to by argp. SPE_NO_CALLBACKS do not use built in library functions.
<i>argp</i>	An (optional) pointer to application specific data, and is passed as the second parameter to the SPE program.
<i>envp</i>	An (optional) pointer to environment specific data, and is passed as the third parameter to the SPE program.

Definition at line 99 of file run.c.

References `__spe_current_active_context`, `_base_spe_handle_library_callback()`, `_base_spe_program_load_complete()`, `spe_context::base_private`, `DEBUG_PRINTF`, `spe_context::base_priv::emulated_entry`, `spe_context::base_priv::entry`, `spe_context::base_priv::fd_spe_dir`, `spe_context::base_priv::flags`, `LS_SIZE`, `spe_context::base_priv::mem_mmap_base`, `spe_context::info::npc`, `spe_context::info::prev`, `spe_stop_info::result`, `spe_stop_info::spe_callback_error`, `SPE_CALLBACK_ERROR`, `SPE_DEFAULT_ENTRY`, `SPE_EVENTS_ENABLE`, `SPE_EXIT`, `spe_stop_info::spe_exit_code`, `spe_context::info::spe_id`, `SPE_ISOLATE`, `SPE_ISOLATE_EMULATE`, `spe_stop_info::spe_isolation_error`, `SPE_ISOLATION_ERROR`, `SPE_NO_CALLBACKS`, `SPE_PROGRAM_ISO_LOAD_COMPLETE`, `SPE_PROGRAM_ISOLATED_STOP`, `SPE_PROGRAM_LIBRARY_CALL`, `SPE_PROGRAM_NORMAL_END`, `SPE_RUN_USER_REGS`, `spe_stop_info::spe_runtime_error`, `SPE_RUNTIME_ERROR`, `spe_stop_info::spe_runtime_exception`, `SPE_RUNTIME_EXCEPTION`, `spe_stop_info::spe_runtime_fatal`, `SPE_RUNTIME_FATAL`, `spe_stop_info::spe_signal_code`, `SPE_SPU_HALT`, `SPE_SPU_INVALID_CHANNEL`, `SPE_SPU_INVALID_INSTR`, `SPE_SPU_STOPPED_BY_STOP`, `SPE_SPU_WAITING_ON_CHANNEL`, `SPE_STOP_AND_SIGNAL`, `spe_stop_info::spu_status`, `spe_context::info::status`, `spe_stop_info::stop_reason`, `addr64::ui`, and `addr64::ull`.

Referenced by `_event_spe_context_run()`.

```

102 {
103     int retval = 0, run_rc;
104     unsigned int run_status, tmp_entry;
105     spe_stop_info_t stopinfo_buf;
106     struct spe_context_info this_context_info __attribute__((cleanup(

```

```

cleanupspeinfo));
107
108     /* If the caller hasn't set a stopinfo buffer, provide a buffer on the
109     * stack instead. */
110     if (!stopinfo)
111         stopinfo = &stopinfo_buf;
112
113
114     /* In emulated isolated mode, the npc will always return as zero.
115     * use our private entry point instead */
116     if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
117         tmp_entry = spe->base_private->emulated_entry;
118
119     else if (*entry == SPE_DEFAULT_ENTRY)
120         tmp_entry = spe->base_private->entry;
121
122     else
123         tmp_entry = *entry;
124
125     /* If we're starting the SPE binary from its original entry point,
126     * setup the arguments to main() */
127     if (tmp_entry == spe->base_private->entry &&
128         !(spe->base_private->flags &
129           (SPE_ISOLATE | SPE_ISOLATE_EMULATE))) {
130
131         addr64 argp64, envp64, tid64, ls64;
132         unsigned int regs[128][4];
133
134         /* setup parameters */
135         argp64.u1l = (uint64_t)(unsigned long)argp;
136         envp64.u1l = (uint64_t)(unsigned long)envp;
137         tid64.u1l = (uint64_t)(unsigned long)spe;
138
139         /* make sure the register values are 0 */
140         memset(regs, 0, sizeof(regs));
141
142         /* set sensible values for stack_ptr and stack_size */
143         regs[1][0] = (unsigned int) LS_SIZE - 16;          /* stack_ptr */
144         regs[2][0] = 0;                                   /* stack_size ( 0 =
145         default ) */
146
147         if (runflags & SPE_RUN_USER_REGS) {
148             /* When SPE_USER_REGS is set, argp points to an array
149             * of 3x128b registers to be passed directly to the SPE
150             * program.
151             */
152             memcpy(regs[3], argp, sizeof(unsigned int) * 12);
153         } else {
154             regs[3][0] = tid64.ui[0];
155             regs[3][1] = tid64.ui[1];
156
157             regs[4][0] = argp64.ui[0];
158             regs[4][1] = argp64.ui[1];
159
160             regs[5][0] = envp64.ui[0];
161             regs[5][1] = envp64.ui[1];
162         }
163
164         /* Store the LS base address in R6 */
165         ls64.u1l = (uint64_t)(unsigned long)spe->base_private->
166         mem_mmap_base;
167         regs[6][0] = ls64.ui[0];
168         regs[6][1] = ls64.ui[1];
169
170         if (set_regs(spe, regs))
171             return -1;
172     }
173
174     /*Leave a trail of breadcrumbs for the debugger to follow */
175     if (!__spe_current_active_context) {
176         __spe_current_active_context = &this_context_info;
177         if (!__spe_current_active_context)
178             return -1;
179         __spe_current_active_context->prev = NULL;
180     } else {
181         struct spe_context_info *newinfo;
182         newinfo = &this_context_info;
183         if (!newinfo)
184             return -1;
185         newinfo->prev = __spe_current_active_context;
186         __spe_current_active_context = newinfo;

```



```

184     }
185     /*remember the ls-addr*/
186     _spe_current_active_context->spe_id = spe->
base_private->fd_spe_dir;
187
188 do_run:
189     /*Remember the npc value*/
190     _spe_current_active_context->npc = tmp_entry;
191
192     /* run SPE context */
193     run_rc = spu_run(spe->base_private->fd_spe_dir,
194                     &tmp_entry, &run_status);
195
196     /*Remember the npc value*/
197     _spe_current_active_context->npc = tmp_entry;
198     _spe_current_active_context->status = run_status;
199
200     DEBUG_PRINTF("spu_run returned run_rc=0x%08x, entry=0x%04x, "
201                 "ext_status=0x%04x.\n", run_rc, tmp_entry, run_status);
202
203     /* set up return values and stopinfo according to spu_run exit
204      * conditions. This is overwritten on error.
205      */
206     stopinfo->spu_status = run_rc;
207
208     if (spe->base_private->flags & SPE_ISOLATE_EMULATE) {
209         /* save the entry point, and pretend that the npc is zero */
210         spe->base_private->emulated_entry = tmp_entry;
211         *entry = 0;
212     } else {
213         *entry = tmp_entry;
214     }
215
216     /* Return with stopinfo set on syscall error paths */
217     if (run_rc == -1) {
218         DEBUG_PRINTF("spu_run returned error %d, errno=%d\n",
219                     run_rc, errno);
220         stopinfo->stop_reason = SPE_RUNTIME_FATAL;
221         stopinfo->result.spe_runtime_fatal = errno;
222         retval = -1;
223
224         /* For isolated contexts, pass EPERM up to the
225          * caller.
226          */
227         if (!(spe->base_private->flags & SPE_ISOLATE
228              && errno == EPERM))
229             errno = EFAULT;
230
231     } else if (run_rc & SPE_SPU_INVALID_INSTR) {
232         DEBUG_PRINTF("SPU has tried to execute an invalid "
233                     "instruction. %d\n", run_rc);
234         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
235         stopinfo->result.spe_runtime_error =
SPE_SPU_INVALID_INSTR;
236         errno = EFAULT;
237         retval = -1;
238
239     } else if ((spe->base_private->flags &
SPE_EVENTS_ENABLE) && run_status) {
240         /* Report asynchronous error if return val are set and
241          * SPU events are enabled.
242          */
243         stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
244         stopinfo->result.spe_runtime_exception = run_status;
245         stopinfo->spu_status = -1;
246         errno = EIO;
247         retval = -1;
248
249     } else if (run_rc & SPE_SPU_STOPPED_BY_STOP) {
250         /* Stop & signals are broken down into three groups
251          * 1. SPE library call
252          * 2. SPE user defined stop & signal
253          * 3. SPE program end.
254          *
255          * These groups are signified by the 14-bit stop code:
256          */
257         int stopcode = (run_rc >> 16) & 0x3fff;
258
259         /* Check if this is a library callback, and callbacks are
260          * allowed (ie, running without SPE_NO_CALLBACKS)

```

```

261         */
262         if ((stopcode & 0xff00) == SPE_PROGRAM_LIBRARY_CALL
263             && !(runflags & SPE_NO_CALLBACKS)) {
264
265             int callback_rc, callback_number = stopcode & 0xff;
266
267             /* execute library callback */
268             DEBUG_PRINTF("SPE library call: %d\n", callback_number);
269             callback_rc = _base_spe_handle_library_callback(
270 spe,
271                                     callback_number, *entry);
272
273             if (callback_rc) {
274                 /* library callback failed; set errno and
275                  * return immediately */
276                 DEBUG_PRINTF("SPE library call failed: %d\n",
277                             callback_rc);
278                 stopinfo->stop_reason =
279 SPE_CALLBACK_ERROR;
280
281                 stopinfo->result.spe_callback_error =
282                     callback_rc;
283                 errno = EFAULT;
284                 retval = -1;
285             } else {
286                 /* successful library callback - restart the SPE
287                  * program at the next instruction */
288                 tmp_entry += 4;
289                 goto do_run;
290             }
291
292             } else if ((stopcode & 0xff00) == SPE_PROGRAM_NORMAL_END) {
293                 /* The SPE program has exited by exit(X) */
294                 stopinfo->stop_reason = SPE_EXIT;
295                 stopinfo->result.spe_exit_code = stopcode & 0xff;
296
297                 if (spe->base_private->flags &
298 SPE_ISOLATE) {
299                     /* Issue an isolated exit, and re-run the SPE.
300                      * We should see a return value without the
301                      * 0x80 bit set. */
302                     if (!issue_isolated_exit(spe))
303                         goto do_run;
304                     retval = -1;
305                 }
306
307             } else if ((stopcode & 0xffff) == SPE_PROGRAM_ISOLATED_STOP) {
308                 /* 0x2206: isolated app has been loaded by loader;
309                  * provide a hook for the debugger to catch this,
310                  * and restart
311                  */
312                 if (stopcode == SPE_PROGRAM_ISO_LOAD_COMPLETE) {
313                     _base_spe_program_load_complete(spe);
314                     goto do_run;
315                 } else {
316                     stopinfo->stop_reason =
317 SPE_ISOLATION_ERROR;
318
319                     stopinfo->result.spe_isolation_error =
320                         stopcode & 0xf;
321                 }
322             } else if (spe->base_private->flags &
323 SPE_ISOLATE &&
324                     !(run_rc & 0x80)) {
325                 /* We've successfully exited isolated mode */
326                 retval = 0;
327             } else {
328                 /* User defined stop & signal, including
329                  * callbacks when disabled */
330                 stopinfo->stop_reason = SPE_STOP_AND_SIGNAL;
331                 stopinfo->result.spe_signal_code = stopcode;
332                 retval = stopcode;
333             }
334
335             } else if (run_rc & SPE_SPU_HALT) {
336                 DEBUG_PRINTF("SPU was stopped by halt. %d\n", run_rc);
337                 stopinfo->stop_reason = SPE_RUNTIME_ERROR;
338                 stopinfo->result.spe_runtime_error =
339 SPE_SPU_HALT;

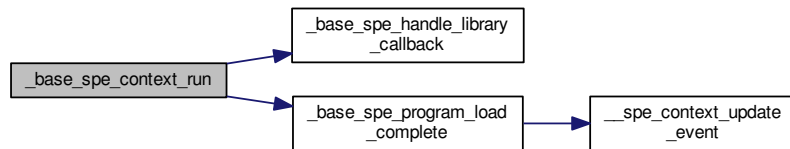
```

```

335         errno = EFAULT;
336         retval = -1;
337
338     } else if (run_rc & SPE_SPU_WAITING_ON_CHANNEL) {
339         DEBUG_PRINTF("SPU is waiting on channel. %d\n", run_rc);
340         stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
341         stopinfo->result.spe_runtime_exception = run_status;
342         stopinfo->spu_status = -1;
343         errno = EIO;
344         retval = -1;
345
346     } else if (run_rc & SPE_SPU_INVALID_CHANNEL) {
347         DEBUG_PRINTF("SPU has tried to access an invalid "
348                     "channel. %d\n", run_rc);
349         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
350         stopinfo->result.spe_runtime_error =
351             SPE_SPU_INVALID_CHANNEL;
352         errno = EFAULT;
353         retval = -1;
354
355     } else {
356         DEBUG_PRINTF("spu.run returned invalid data: 0x%04x\n", run_rc);
357         stopinfo->stop_reason = SPE_RUNTIME_FATAL;
358         stopinfo->result.spe_runtime_fatal = -1;
359         stopinfo->spu_status = -1;
360         errno = EFAULT;
361         retval = -1;
362     }
363
364     freespeinfo();
365     return retval;
366 }

```

Here is the call graph for this function:



3.21.3 Variable Documentation

`__thread struct spe_context_info* __spe_current_active_context`

Referenced by `_base_spe_context_run()`.

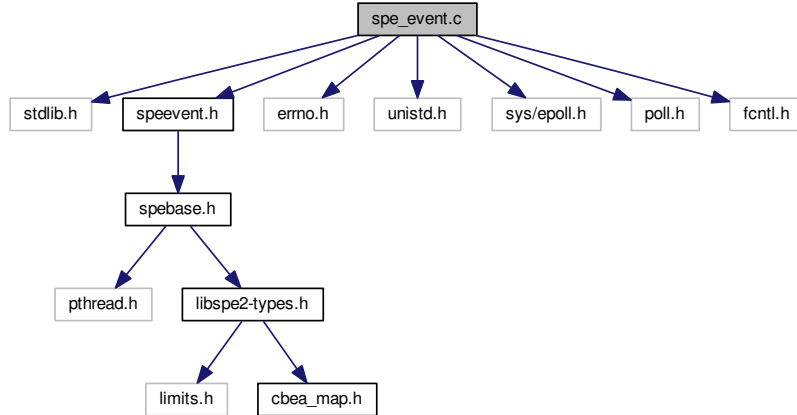
3.22 spe_event.c File Reference

```

#include <stdlib.h>
#include "speevent.h"
#include <errno.h>
#include <unistd.h>
#include <sys/epoll.h>
#include <poll.h>
#include <fcntl.h>

```

Include dependency graph for `spe_event.c`:



Macros

- `#define __SPE_EVENT_ALL`
- `#define __SPE_EPOLL_SIZE 10`
- `#define __SPE_EPOLL_FD_GET(handler) (*(int*)(handler))`
- `#define __SPE_EPOLL_FD_SET(handler, fd) (*(int*)(handler) = (fd))`
- `#define __SPE_EVENT_CONTEXT_PRIV_GET(spe) ((spe_context_event_priv_ptr_t)(spe)->event_private)`
- `#define __SPE_EVENT_CONTEXT_PRIV_SET(spe, evctx) ((spe)->event_private = (evctx))`
- `#define __SPE_EVENTS_ENABLED(spe) ((spe)->base_private->flags & SPE_EVENTS_ENABLED)`

Functions

- `void _event_spe_context_lock (spe_context_ptr_t spe)`
- `void _event_spe_context_unlock (spe_context_ptr_t spe)`
- `int _event_spe_stop_info_read (spe_context_ptr_t spe, spe_stop_info_t *stopinfo)`
- `spe_event_handler_ptr_t _event_spe_event_handler_create (void)`
- `int _event_spe_event_handler_destroy (spe_event_handler_ptr_t evhandler)`
- `int _event_spe_event_handler_register (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)`
- `int _event_spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)`
- `int _event_spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t *events, int max_events, int timeout)`
- `int _event_spe_context_finalize (spe_context_ptr_t spe)`
- `struct spe_context_event_priv * _event_spe_context_initialize (spe_context_ptr_t spe)`
- `int _event_spe_context_run (spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)`

3.22.1 Macro Definition Documentation

#define __SPE_EPOLL_FD_GET(*handler*) (*(int*)(handler))

Definition at line 37 of file spe_event.c.

Referenced by `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_destroy()`, `_event_spe_event_handler_register()`, and `_event_spe_event_wait()`.

#define __SPE_EPOLL_FD_SET(*handler, fd*) (*(int*)(handler) = (fd))

Definition at line 38 of file spe_event.c.

Referenced by `_event_spe_event_handler_create()`.

#define __SPE_EPOLL_SIZE 10

Definition at line 35 of file spe_event.c.

Referenced by `_event_spe_event_handler_create()`.

#define __SPE_EVENT_ALL

Value:

```
( SPE_EVENT_OUT_INTR_MBOX | SPE_EVENT_IN_MBOX |
  \
  SPE_EVENT_TAG_GROUP | SPE_EVENT_SPE_STOPPED )
```

Definition at line 31 of file spe_event.c.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

#define __SPE_EVENT_CONTEXT_PRIV_GET(*spe*) ((spe->context_event_priv_ptr_t)(spe)->event_private)

Definition at line 40 of file spe_event.c.

Referenced by `_event_spe_context_finalize()`, `_event_spe_context_lock()`, `_event_spe_context_run()`, `_event_spe_context_unlock()`, `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_register()`, and `_event_spe_stop_info_read()`.

#define __SPE_EVENT_CONTEXT_PRIV_SET(*spe, evctx*) ((spe->event_private = (evctx))

Definition at line 42 of file spe_event.c.

Referenced by `_event_spe_context_finalize()`.

#define __SPE_EVENTS_ENABLED(*spe*) ((spe->base_private->flags & SPE_EVENTS_ENABLE)

Definition at line 45 of file spe_event.c.

Referenced by `_event_spe_event_handler_deregister()`, and `_event_spe_event_handler_register()`.

3.22.2 Function Documentation

int `_event_spe_context_finalize` (*spe_context_ptr_t spe*)

Definition at line 416 of file spe_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_CONTEXT_PRIV_SET`, `spe_context->event_priv::lock`, `spe_context.event_priv::stop_event_pipe`, and `spe_context.event_priv::stop_event_read_lock`.

```

417 {
418     spe_context_event_priv_ptr_t evctx;
419
420     if (!spe) {
421         errno = ESRCH;
422         return -1;
423     }
424
425     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
426     __SPE_EVENT_CONTEXT_PRIV_SET(spe, NULL);
427
428     close(evctx->stop_event_pipe[0]);
429     close(evctx->stop_event_pipe[1]);
430
431     pthread_mutex_destroy(&evctx->lock);
432     pthread_mutex_destroy(&evctx->stop_event_read_lock);
433
434     free(evctx);
435
436     return 0;
437 }

```

struct spe_context_event_priv* _event_spe_context_initialize (spe_context_ptr_t spe)

Definition at line 439 of file spe_event.c.

References spe_context_event_priv::events, spe_context_event_priv::lock, spe_event_unit::spe, spe_context_event_priv::stop_event_pipe, and spe_context_event_priv::stop_event_read_lock.

```

440 {
441     spe_context_event_priv_ptr_t evctx;
442     int rc;
443     int i;
444
445     evctx = calloc(1, sizeof(*evctx));
446     if (!evctx) {
447         return NULL;
448     }
449
450     rc = pipe(evctx->stop_event_pipe);
451     if (rc == -1) {
452         free(evctx);
453         return NULL;
454     }
455     rc = fcntl(evctx->stop_event_pipe[0], F_GETFL);
456     if (rc != -1) {
457         rc = fcntl(evctx->stop_event_pipe[0], F_SETFL, rc | O_NONBLOCK);
458     }
459     if (rc == -1) {
460         close(evctx->stop_event_pipe[0]);
461         close(evctx->stop_event_pipe[1]);
462         free(evctx);
463         errno = EIO;
464         return NULL;
465     }
466
467     for (i = 0; i < sizeof(evctx->events) / sizeof(evctx->events[0]); i++) {
468         evctx->events[i].spe = spe;
469     }
470
471     pthread_mutex_init(&evctx->lock, NULL);
472     pthread_mutex_init(&evctx->stop_event_read_lock, NULL);
473
474     return evctx;
475 }

```

void _event_spe_context_lock (spe_context_ptr_t spe)

Definition at line 49 of file spe_event.c.

References __SPE_EVENT_CONTEXT_PRIV_GET.

Referenced by _event_spe_event_handler_deregister(), _event_spe_event_handler_register(), and _event_spe_event_wait().

```

50 {
51     pthread_mutex_lock(&__SPE_EVENT_CONTEXT_PRIV_GET(spe)->lock);
52 }

```

int _event_spe_context_run (spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp, void * envp, spe_stop_info_t * stopinfo)

Definition at line 477 of file spe_event.c.

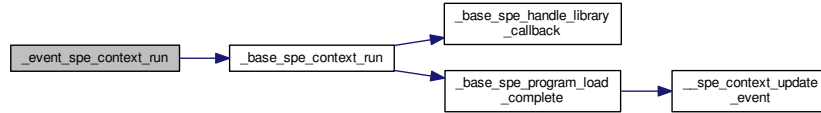
References __SPE_EVENT_CONTEXT_PRIV_GET, _base_spe_context_run(), and spe_context_event_priv::stop_event_pipe.

```

478 {
479     spe_context_event_priv_ptr_t evctx;
480     spe_stop_info_t stopinfo_buf;
481     int rc;
482
483     if (!stopinfo) {
484         stopinfo = &stopinfo_buf;
485     }
486     rc = _base_spe_context_run(spe, entry, runflags, argp, envp, stopinfo);
487
488     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
489     if (write(evctx->stop_event_pipe[1], stopinfo, sizeof(*stopinfo)) != sizeof(*stopinfo)) {
490         /* error check. */
491     }
492
493     return rc;
494 }

```

Here is the call graph for this function:



void _event_spe_context_unlock (spe_context_ptr_t spe)

Definition at line 54 of file spe_event.c.

References __SPE_EVENT_CONTEXT_PRIV_GET.

Referenced by _event_spe_event_handler_deregister(), _event_spe_event_handler_register(), and _event_spe_event_wait().

```

55 {
56     pthread_mutex_unlock(&__SPE_EVENT_CONTEXT_PRIV_GET(spe)->lock);
57 }

```

spe_event_handler_ptr_t _event_spe_event_handler_create (void)

Definition at line 110 of file spe_event.c.

References __SPE_EPOLL_FD_SET, and __SPE_EPOLL_SIZE.

```

111 {
112     int epfd;
113     spe_event_handler_t *evhandler;
114
115     evhandler = calloc(1, sizeof(*evhandler));
116     if (!evhandler) {

```

```

117     return NULL;
118 }
119
120 epfd = epoll_create(__SPE_EPOLL_SIZE);
121 if (epfd == -1) {
122     free(evhandler);
123     return NULL;
124 }
125
126 __SPE_EPOLL_FD_SET(evhandler, epfd);
127
128 return evhandler;
129 }

```

int _event_spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t * event)

Definition at line 273 of file spe_event.c.

References `__base_spe_event_source_acquire()`, `__SPE_EPOLL_FD_GET`, `__SPE_EVENT_ALL`, `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_IN_MBOX`, `__SPE_EVENT_OUT_INTR_MBOX`, `__SPE_EVENT_SPE_STOPPED`, `__SPE_EVENT_TAG_GROUP`, `__SPE_EVENTS_ENABLED`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, `spe_context_event_priv::events`, `spe_event_unit::events`, `FD_IBOX`, `FD_MFC`, `FD_WBOX`, `spe_event_unit::spe`, `SPE_EVENT_IN_MBOX`, `SPE_EVENT_OUT_INTR_MBOX`, `SPE_EVENT_SPE_STOPPED`, `SPE_EVENT_TAG_GROUP`, and `spe_context_event_priv::stop_event_pipe`.

```

274 {
275     int epfd;
276     const int ep_op = EPOLL_CTL_DEL;
277     spe_context_event_priv_ptr_t evctx;
278     int fd;
279
280     if (!evhandler) {
281         errno = ESRCH;
282         return -1;
283     }
284     if (!event || !event->spe) {
285         errno = EINVAL;
286         return -1;
287     }
288     if (!__SPE_EVENTS_ENABLED(event->spe)) {
289         errno = ENOTSUP;
290         return -1;
291     }
292
293     epfd = __SPE_EPOLL_FD_GET(evhandler);
294     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);
295
296     if (event->events & ~__SPE_EVENT_ALL) {
297         errno = ENOTSUP;
298         return -1;
299     }
300
301     _event_spe_context_lock(event->spe); /* for spe->event.private->events */
302
303     if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
304         fd = __base_spe_event_source_acquire(event->
305         spe, FD_IBOX);
306         if (fd == -1) {
307             _event_spe_context_unlock(event->spe);
308             return -1;
309         }
310         if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
311             _event_spe_context_unlock(event->spe);
312             return -1;
313         }
314         evctx->events[__SPE_EVENT_OUT_INTR_MBOX].
315         events = 0;
316     }
317
318     if (event->events & SPE_EVENT_IN_MBOX) {
319         fd = __base_spe_event_source_acquire(event->
320         spe, FD_WBOX);
321         if (fd == -1) {
322             _event_spe_context_unlock(event->spe);

```

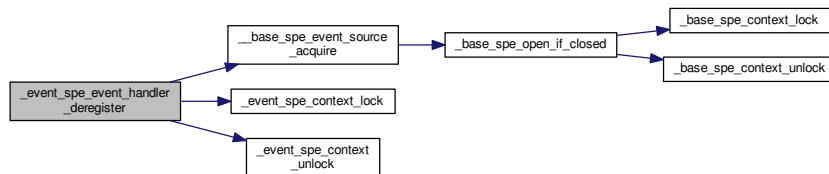


```

320     return -1;
321 }
322 if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
323     _event_spe_context_unlock(event->spe);
324     return -1;
325 }
326 evctx->events[_SPE_EVENT_IN_MBOX].events = 0;
327 }
328
329 if (event->events & SPE_EVENT_TAG_GROUP) {
330     fd = _base_spe_event_source_acquire(event->
spe, FD_MFC);
331     if (fd == -1) {
332         _event_spe_context_unlock(event->spe);
333         return -1;
334     }
335     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
336         _event_spe_context_unlock(event->spe);
337         return -1;
338     }
339     evctx->events[_SPE_EVENT_TAG_GROUP].events = 0;
340 }
341
342 if (event->events & SPE_EVENT_SPE_STOPPED) {
343     fd = evctx->stop_event_pipe[0];
344     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
345         _event_spe_context_unlock(event->spe);
346         return -1;
347     }
348     evctx->events[_SPE_EVENT_SPE_STOPPED].events = 0;
349 }
350
351 _event_spe_context_unlock(event->spe);
352
353 return 0;
354 }

```

Here is the call graph for this function:



int _event_spe_event_handler_destroy (spe_event_handler_ptr_t evhandler)

Definition at line 135 of file spe_event.c.

References `__SPE_EPOLL_FD_GET`.

```

136 {
137     int epfd;
138
139     if (!evhandler) {
140         errno = ESRCH;
141         return -1;
142     }
143
144     epfd = __SPE_EPOLL_FD_GET(evhandler);
145     close(epfd);
146
147     free(evhandler);
148     return 0;
149 }

```

int _event_spe_event_handler_register (spe_event_handler_ptr_t evhandler, spe_event_unit_t * event)

Definition at line 155 of file spe_event.c.

References `__base_spe_event_source_acquire()`, `__SPE_EPOLL_FD_GET`, `__SPE_EVENT_ALL`, `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_IN_MBOX`, `__SPE_EVENT_OUT_INTR_MBOX`, `__SPE_EVENT_SPE_STOPPED`, `__SPE_EVENT_TAG_GROUP`, `__SPE_EVENTS_ENABLED`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, `spe_context::base_private`, `spe_event_unit::data`, `spe_context_event_priv::events`, `spe_event_unit::events`, `FD_IBOX`, `FD_MFC`, `FD_WBOX`, `spe_context_base_priv::flags`, `spe_event_data::ptr`, `spe_event_unit::spe`, `SPE_EVENT_IN_MBOX`, `SPE_EVENT_OUT_INTR_MBOX`, `SPE_EVENT_SPE_STOPPED`, `SPE_EVENT_TAG_GROUP`, `SPE_MAP_PS`, and `spe_context_event_priv::stop_event_pipe`.

```

156 {
157     int epfd;
158     const int ep_op = EPOLL_CTL_ADD;
159     spe_context_event_priv_ptr_t evctx;
160     spe_event_unit_t *ev_buf;
161     struct epoll_event ep_event;
162     int fd;
163
164     if (!evhandler) {
165         errno = ESRCH;
166         return -1;
167     }
168     if (!event || !event->spe) {
169         errno = EINVAL;
170         return -1;
171     }
172     if (!__SPE_EVENTS_ENABLED(event->spe)) {
173         errno = ENOTSUP;
174         return -1;
175     }
176
177     epfd = __SPE_EPOLL_FD_GET(evhandler);
178     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);
179
180     if (event->events & ~__SPE_EVENT_ALL) {
181         errno = ENOTSUP;
182         return -1;
183     }
184
185     _event_spe_context_lock(event->spe); /* for spe->event_private->events */
186
187     if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
188         fd = __base_spe_event_source_acquire(event->
189         spe, FD_IBOX);
190         if (fd == -1) {
191             _event_spe_context_unlock(event->spe);
192             return -1;
193         }
194         ev_buf = &evctx->events[__SPE_EVENT_OUT_INTR_MBOX];
195         ev_buf->events = SPE_EVENT_OUT_INTR_MBOX;
196         ev_buf->data = event->data;
197
198         ep_event.events = EPOLLIN;
199         ep_event.data.ptr = ev_buf;
200         if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
201             _event_spe_context_unlock(event->spe);
202             return -1;
203         }
204     }
205
206     if (event->events & SPE_EVENT_IN_MBOX) {
207         fd = __base_spe_event_source_acquire(event->
208         spe, FD_WBOX);
209         if (fd == -1) {
210             _event_spe_context_unlock(event->spe);
211             return -1;
212         }
213         ev_buf = &evctx->events[__SPE_EVENT_IN_MBOX];
214         ev_buf->events = SPE_EVENT_IN_MBOX;
215         ev_buf->data = event->data;
216
217         ep_event.events = EPOLLOUT;

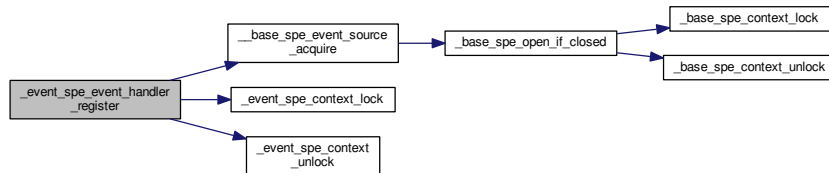
```

```

218     ep.event.data.ptr = ev_buf;
219     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
220         _event_spe_context_unlock(event->spe);
221         return -1;
222     }
223 }
224
225 if (event->events & SPE_EVENT_TAG_GROUP) {
226     fd = __base_spe_event_source_acquire(event->
spe, FD_MFC);
227     if (fd == -1) {
228         _event_spe_context_unlock(event->spe);
229         return -1;
230     }
231
232     if (event->spe->base.private->flags & SPE_MAP_PS) {
233         _event_spe_context_unlock(event->spe);
234         errno = ENOTSUP;
235         return -1;
236     }
237
238     ev_buf = &evctx->events[_SPE_EVENT_TAG_GROUP];
239     ev_buf->events = SPE_EVENT_TAG_GROUP;
240     ev_buf->data = event->data;
241
242     ep_event.events = EPOLLIN;
243     ep_event.data.ptr = ev_buf;
244     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
245         _event_spe_context_unlock(event->spe);
246         return -1;
247     }
248 }
249
250 if (event->events & SPE_EVENT_SPE_STOPPED) {
251     fd = evctx->stop_event_pipe[0];
252
253     ev_buf = &evctx->events[_SPE_EVENT_SPE_STOPPED];
254     ev_buf->events = SPE_EVENT_SPE_STOPPED;
255     ev_buf->data = event->data;
256
257     ep_event.events = EPOLLIN;
258     ep_event.data.ptr = ev_buf;
259     if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
260         _event_spe_context_unlock(event->spe);
261         return -1;
262     }
263 }
264
265 _event_spe_context_unlock(event->spe);
266
267 return 0;
268 }

```

Here is the call graph for this function:



int _event_spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t * events, int max_events, int timeout)

Definition at line 360 of file spe_event.c.

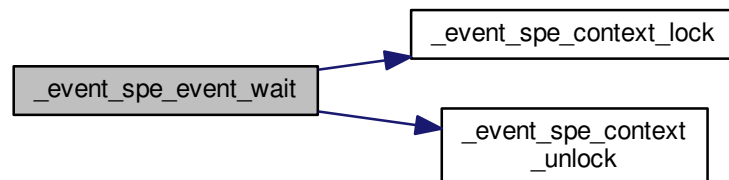
References `__SPE_EPOLL_FD_GET`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, and `spe_event_unit::spe`.

```

361 {
362     int epfd;
363     struct epoll_event *ep_events;
364     int rc;
365
366     if (!evhandler) {
367         errno = ESRCH;
368         return -1;
369     }
370     if (!events || max_events <= 0) {
371         errno = EINVAL;
372         return -1;
373     }
374
375     epfd = __SPE_EPOLL_FD_GET(evhandler);
376
377     ep_events = malloc(sizeof(*ep_events) * max_events);
378     if (!ep_events) {
379         return -1;
380     }
381
382     for ( ; ; ) {
383         rc = epoll_wait(epfd, ep_events, max_events, timeout);
384         if (rc == -1) { /* error */
385             if (errno == EINTR) {
386                 if (timeout >= 0) { /* behave as timeout */
387                     rc = 0;
388                     break;
389                 }
390                 /* else retry */
391             }
392             else {
393                 break;
394             }
395         }
396         else if (rc > 0) {
397             int i;
398             for (i = 0; i < rc; i++) {
399                 spe_event_unit_t *ev = (spe_event_unit_t *) (ep_events[i].data.ptr);
400                 _event_spe_context_lock(ev->spe); /* lock ev itself */
401                 events[i] = *ev;
402                 _event_spe_context_unlock(ev->spe);
403             }
404             break;
405         }
406         else { /* timeout */
407             break;
408         }
409     }
410
411     free(ep_events);
412
413     return rc;
414 }

```

Here is the call graph for this function:



int _event_spe_stop_info_read (spe_context_ptr_t spe, spe_stop_info_t * stopinfo)

Definition at line 59 of file spe_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`, `spe_context_event_priv::stop_event_pipe`, and `spe_context_event_priv::stop_event_read_lock`.

```

60 {
61     spe_context_event_priv_ptr_t evctx;
62     int rc;
63     int fd;
64     size_t total;
65
66     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
67     fd = evctx->stop_event_pipe[0];
68
69     pthread_mutex_lock(&evctx->stop_event_read_lock); /* for atomic read */
70
71     rc = read(fd, stopinfo, sizeof(*stopinfo));
72     if (rc == -1) {
73         pthread_mutex_unlock(&evctx->stop_event_read_lock);
74         return -1;
75     }
76
77     total = rc;
78     while (total < sizeof(*stopinfo)) { /* this loop will be executed in few cases */
79         struct pollfd fds;
80         fds.fd = fd;
81         fds.events = POLLIN;
82         rc = poll(&fds, 1, -1);
83         if (rc == -1) {
84             if (errno != EINTR) {
85                 break;
86             }
87         }
88         else if (rc == 1) {
89             rc = read(fd, (char *)stopinfo + total, sizeof(*stopinfo) - total);
90             if (rc == -1) {
91                 if (errno != EAGAIN) {
92                     break;
93                 }
94             }
95             else {
96                 total += rc;
97             }
98         }
99     }
100
101     pthread_mutex_unlock(&evctx->stop_event_read_lock);
102
103     return rc == -1 ? -1 : 0;
104 }

```

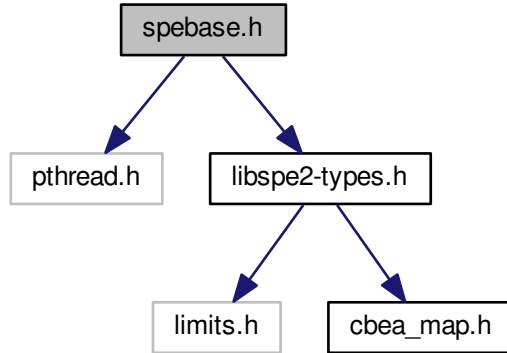
3.23 spebase.h File Reference

```

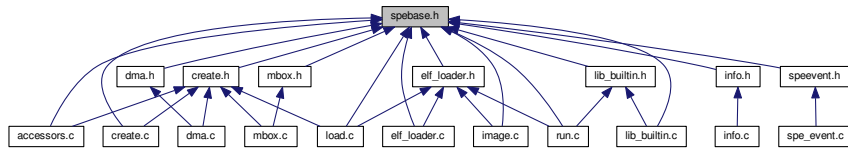
#include <pthread.h>
#include "libspe2-types.h"

```

Include dependency graph for `spebase.h`:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct `spe_context_base_priv`
- struct `spe_gang_context_base_priv`

Macros

- `#define __PRINTF(fmt, args...) { fprintf(stderr,fmt , ## args); }`
- `#define DEBUG_PRINTF(fmt, args...)`
- `#define LS_SIZE 0x40000 /* 256K (in bytes) */`
- `#define PSMAP_SIZE 0x20000 /* 128K (in bytes) */`
- `#define MFC_SIZE 0x1000`
- `#define MSS_SIZE 0x1000`
- `#define CNTL_SIZE 0x1000`
- `#define SIGNAL_SIZE 0x1000`
- `#define MSSYNC_OFFSET 0x00000`
- `#define MFC_OFFSET 0x03000`
- `#define CNTL_OFFSET 0x04000`
- `#define SIGNAL1_OFFSET 0x14000`
- `#define SIGNAL2_OFFSET 0x1c000`

- #define `SPE_EMULATE_PARAM_BUFFER` 0x3e000
- #define `SPE_PROGRAM_NORMAL_END` 0x2000
- #define `SPE_PROGRAM_LIBRARY_CALL` 0x2100
- #define `SPE_PROGRAM_ISOLATED_STOP` 0x2200
- #define `SPE_PROGRAM_ISO_LOAD_COMPLETE` 0x2206

Enumerations

- enum `fd_name` {
`FD_MBOX`, `FD_MBOX_STAT`, `FD_IBOX`, `FD_IBOX_NB`,
`FD_IBOX_STAT`, `FD_WBOX`, `FD_WBOX_NB`, `FD_WBOX_STAT`,
`FD_SIG1`, `FD_SIG2`, `FD_MFC`, `FD_MSS`,
`NUM_MBOX_FDS` }

Functions

- `spe_context_ptr_t _base_spe_context_create` (unsigned int flags, `spe_gang_context_ptr_t` gctx, `spe_context_ptr_t` aff_spe)
- `spe_gang_context_ptr_t _base_spe_gang_context_create` (unsigned int flags)
- `int _base_spe_program_load` (`spe_context_ptr_t` spectx, `spe_program_handle_t` *program)
- `void _base_spe_program_load_complete` (`spe_context_ptr_t` spectx)
- `int _base_spe_emulated_loader_present` (void)
- `int _base_spe_context_destroy` (`spe_context_ptr_t` spectx)
- `int _base_spe_gang_context_destroy` (`spe_gang_context_ptr_t` gctx)
- `int _base_spe_context_run` (`spe_context_ptr_t` spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, `spe_stop_info_t` *stopinfo)
- `int _base_spe_image_close` (`spe_program_handle_t` *handle)
- `spe_program_handle_t * _base_spe_image_open` (const char *filename)
- `int _base_spe_mfcio_put` (`spe_context_ptr_t` spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int _base_spe_mfcio_putb` (`spe_context_ptr_t` spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int _base_spe_mfcio_putf` (`spe_context_ptr_t` spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int _base_spe_mfcio_get` (`spe_context_ptr_t` spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int _base_spe_mfcio_getb` (`spe_context_ptr_t` spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int _base_spe_mfcio_getf` (`spe_context_ptr_t` spectx, unsigned int ls, void *ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)
- `int _base_spe_out_mbox_read` (`spe_context_ptr_t` spectx, unsigned int mbox_data[], int count)
- `int _base_spe_in_mbox_write` (`spe_context_ptr_t` spectx, unsigned int mbox_data[], int count, int behavior_flag)
- `int _base_spe_in_mbox_status` (`spe_context_ptr_t` spectx)
- `int _base_spe_out_mbox_status` (`spe_context_ptr_t` spectx)
- `int _base_spe_out_intr_mbox_status` (`spe_context_ptr_t` spectx)
- `int _base_spe_out_intr_mbox_read` (`spe_context_ptr_t` spectx, unsigned int mbox_data[], int count, int behavior_flag)
- `int _base_spe_signal_write` (`spe_context_ptr_t` spectx, unsigned int signal_reg, unsigned int data)
- `int _base_spe_callback_handler_register` (void *handler, unsigned int callnum, unsigned int mode)
- `int _base_spe_callback_handler_deregister` (unsigned int callnum)
- `void * _base_spe_callback_handler_query` (unsigned int callnum)

- `int _base_spe_stop_reason_get (spe_context_ptr_t spectx)`
- `int _base_spe_mfcio_tag_status_read (spe_context_ptr_t spectx, unsigned int mask, unsigned int behavior, unsigned int *tag_status)`
- `int _base_spe_stop_event_source_get (spe_context_ptr_t spectx)`
- `int _base_spe_stop_event_target_get (spe_context_ptr_t spectx)`
- `int _base_spe_stop_status_get (spe_context_ptr_t spectx)`
- `int _base_spe_event_source_acquire (struct spe_context *spectx, enum fd_name fdesc)`
- `void _base_spe_event_source_release (struct spe_context *spectx, enum fd_name fdesc)`
- `void * _base_spe_ps_area_get (struct spe_context *spectx, enum ps_area area)`
- `int _base_spe_spe_dir_get (struct spe_context *spectx)`
- `void * _base_spe_ls_area_get (struct spe_context *spectx)`
- `int _base_spe_ls_size_get (spe_context_ptr_t spe)`
- `void _base_spe_context_lock (spe_context_ptr_t spe, enum fd_name fd)`
- `void _base_spe_context_unlock (spe_context_ptr_t spe, enum fd_name fd)`
- `int _base_spe_cpu_info_get (int info_requested, int cpu_node)`
- `void __spe_context_update_event (void)`
- `int _base_spe_mssync_start (spe_context_ptr_t spectx)`
- `int _base_spe_mssync_status (spe_context_ptr_t spectx)`

3.23.1 Detailed Description

[spebase.h](#) contains the public API functions

Definition in file [spebase.h](#).

3.23.2 Macro Definition Documentation

```
#define __PRINTF( fmt, args... ) { fprintf(stderr,fmt , ## args); }
```

Definition at line 34 of file [spebase.h](#).

```
#define CNTL_OFFSET 0x04000
```

Definition at line 124 of file [spebase.h](#).

Referenced by [_base_spe_context.create\(\)](#).

```
#define CNTL_SIZE 0x1000
```

Definition at line 119 of file [spebase.h](#).

Referenced by [_base_spe_context.create\(\)](#).

```
#define DEBUG_PRINTF( fmt, args... )
```

Definition at line 38 of file [spebase.h](#).

```
#define LS_SIZE 0x40000 /* 256K (in bytes) */
```

Definition at line 115 of file [spebase.h](#).

```
#define MFC_OFFSET 0x03000
```

Definition at line 123 of file [spebase.h](#).

Referenced by [_base_spe_context.create\(\)](#).

#define MFC_SIZE 0x1000

Definition at line 117 of file spebase.h.

Referenced by `_base_spe_context_create()`.

#define MSS_SIZE 0x1000

Definition at line 118 of file spebase.h.

Referenced by `_base_spe_context_create()`.

#define MSSYNC_OFFSET 0x00000

Definition at line 122 of file spebase.h.

Referenced by `_base_spe_context_create()`.

#define PSMAP_SIZE 0x20000 /* 128K (in bytes) */

Definition at line 116 of file spebase.h.

Referenced by `_base_spe_context_create()`.

#define SIGNAL1_OFFSET 0x14000

Definition at line 125 of file spebase.h.

Referenced by `_base_spe_context_create()`.

#define SIGNAL2_OFFSET 0x1c000

Definition at line 126 of file spebase.h.

Referenced by `_base_spe_context_create()`.

#define SIGNAL_SIZE 0x1000

Definition at line 120 of file spebase.h.

Referenced by `_base_spe_context_create()`.

#define SPE_EMULATE_PARAM_BUFFER 0x3e000

Location of the PPE-assisted library call buffer for emulated isolation contexts.

Definition at line 132 of file spebase.h.

Referenced by `_base_spe_handle_library_callback()`.

#define SPE_PROGRAM_ISO_LOAD_COMPLETE 0x2206

Definition at line 143 of file spebase.h.

Referenced by `_base_spe_context_run()`.

#define SPE_PROGRAM_ISOLATED_STOP 0x2200

Isolated exit codes: 0x220x

Definition at line 142 of file spebase.h.

Referenced by `_base_spe_context_run()`.

#define SPE_PROGRAM_LIBRARY_CALL 0x2100

Definition at line 137 of file spebase.h.

Referenced by `_base_spe_context_run()`.

#define SPE_PROGRAM_NORMAL_END 0x2000

Definition at line 136 of file spebase.h.

Referenced by `_base_spe_context_run()`.

3.23.3 Enumeration Type Documentation

enum fd_name

NOTE: NUM_MBOX_FDS must always be the last element in the enumeration

Enumerator

FD_MBOX
FD_MBOX_STAT
FD_IBOX
FD_IBOX_NB
FD_IBOX_STAT
FD_WBOX
FD_WBOX_NB
FD_WBOX_STAT
FD_SIG1
FD_SIG2
FD_MFC
FD_MSS
NUM_MBOX_FDS

Definition at line 42 of file spebase.h.

```

42     {
43         FD_MBOX,
44         FD_MBOX_STAT,
45         FD_IBOX,
46         FD_IBOX_NB,
47         FD_IBOX_STAT,
48         FD_WBOX,
49         FD_WBOX_NB,
50         FD_WBOX_STAT,
51         FD_SIG1,
52         FD_SIG2,
53         FD_MFC,
54         FD_MSS,
55         NUM_MBOX_FDS
56     };

```

3.23.4 Function Documentation

int `_base_spe_event_source_acquire` (struct spe_context * *spectx*, enum fd_name *fdesc*)

`_base_spe_event_source_acquire` opens a file descriptor to the specified event source

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

<i>fdesc</i>	Specifies the event source
--------------	----------------------------

void __base_spe_event_source_release (struct spe_context * *spectx*, enum fd_name *fdesc*)

__base_spe_event_source_release releases the file descriptor to the specified event source

Parameters

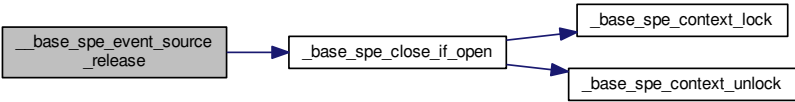
<i>spectx</i>	Specifies the SPE context
<i>fdesc</i>	Specifies the event source

Definition at line 79 of file accessors.c.

References `_base_spe_close_if_open()`.

```
80 {  
81     _base_spe_close_if_open(spe, fdesc);  
82 }
```

Here is the call graph for this function:



int __base_spe_spe_dir_get (struct spe_context * *spectx*)

__base_spe_spe_dir_get return the file descriptor of the SPE directory in spufs

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

int __base_spe_stop_event_source_get (spe_context_ptr_t *spe*)

__base_spe_stop_event_source_get

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

speevent users read from this end

Definition at line 92 of file accessors.c.

References `spe_context::base_private`, and `spe_context_base_priv::ev_pipe`.

```
93 {  
94     return spe->base_private->ev_pipe[1];  
95 }
```

int __base_spe_stop_event_target_get (spe_context_ptr_t *spe*)

__base_spe_stop_event_target_get

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

speevent writes to this end

Definition at line 100 of file accessors.c.

References spe_context::base_private, and spe_context_base_priv::ev_pipe.

```

101 {
102     return spe->base_private->ev_pipe[0];
103 }
```

void __spe_context_update_event (void)

__spe_context_update_event internal function for gdb notification.

Referenced by _base_spe_context_destroy(), and _base_spe_program_load_complete().

int _base_spe_callback_handler_deregister (unsigned int *callnum*)

unregister a handler function for the specified number NOTE: unregistering a handler from call zero and one is ignored.

Definition at line 78 of file lib_builtin.c.

References MAX_CALLNUM, and RESERVED.

```

79 {
80     errno = 0;
81     if (callnum > MAX_CALLNUM) {
82         errno = EINVAL;
83         return -1;
84     }
85     if (callnum < RESERVED) {
86         errno = EACCES;
87         return -1;
88     }
89     if (handlers[callnum] == NULL) {
90         errno = ESRCH;
91         return -1;
92     }
93
94     handlers[callnum] = NULL;
95     return 0;
96 }
```

void* _base_spe_callback_handler_query (unsigned int *callnum*)

query a handler function for the specified number

Definition at line 98 of file lib_builtin.c.

References MAX_CALLNUM.

```

99 {
100     errno = 0;
101
102     if (callnum > MAX_CALLNUM) {
103         errno = EINVAL;
104         return NULL;
105     }
106     if (handlers[callnum] == NULL) {
107         errno = ESRCH;
108         return NULL;
109     }
110     return handlers[callnum];
111 }
```

int _base_spe_callback_handler_register (void * *handler*, unsigned int *callnum*, unsigned int *mode*)

register a handler function for the specified number NOTE: registering a handler to call zero and one is ignored.

Definition at line 40 of file lib.builtin.c.

References MAX_CALLNUM, RESERVED, SPE_CALLBACK_NEW, and SPE_CALLBACK_UPDATE.

```

41 {
42     errno = 0;
43
44     if (callnum > MAX_CALLNUM) {
45         errno = EINVAL;
46         return -1;
47     }
48
49     switch(mode) {
50     case SPE_CALLBACK_NEW:
51         if (callnum < RESERVED) {
52             errno = EACCES;
53             return -1;
54         }
55         if (handlers[callnum] != NULL) {
56             errno = EACCES;
57             return -1;
58         }
59         handlers[callnum] = handler;
60         break;
61
62     case SPE_CALLBACK_UPDATE:
63         if (handlers[callnum] == NULL) {
64             errno = ESRCH;
65             return -1;
66         }
67         handlers[callnum] = handler;
68         break;
69     default:
70         errno = EINVAL;
71         return -1;
72         break;
73     }
74     return 0;
75 }
76 }
```

spe_context_ptr_t _base_spe_context_create (unsigned int *flags*, spe_gang_context_ptr_t *gctx*, spe_context_ptr_t *aff_spe*)

_base_spe_context_create creates a single SPE context, i.e., the corresponding directory is created in SPUFS either as a subdirectory of a gang or individually (maybe this is best considered a gang of one)

Parameters

<i>flags</i>	
<i>gctx</i>	specify NULL if not belonging to a gang
<i>aff_spe</i>	specify NULL to skip affinity information

Definition at line 183 of file create.c.

References _base_spe_emulated_loader_present(), spe_context::base_private, spe_gang_context::base_private, spe_context_base_priv::cntl_mmap_base, CNTL_OFFSET, CNTL_SIZE, DEBUG_PRINTF, spe_context_base_priv::fd_lock, spe_context_base_priv::fd_spe_dir, spe_context_base_priv::flags, spe_gang_context_base_priv::gangname, spe_context_base_priv::loaded_program, LS_SIZE, spe_context_base_priv::mem_mmap_base, spe_context_base_priv::mfc_mmap_base, MFC_OFFSET, MFC_SIZE, MSS_SIZE, spe_context_base_priv::mssync_mmap_base, MSSYNC_OFFSET, NUM_MBOX_FDS, spe_context_base_priv::psmap_mmap_base, PSMAP_SIZE, spe_context_base_priv::signal1_mmap_base, SIGNAL1_OFFSET, spe_context_base_priv::signal2_mmap_base, SIGNAL2_OFFSET, SIGNAL_SIZE, SPE_AFFINITY_MEMORY, SPE_CFG_SIGNOTIFY1_OR, SPE_CFG_SIGNOTIFY2_OR, SPE_EVENTS_ENABLE, spe_context_base_priv::spe_fds_array, SPE_ISOLATE, SPE_ISOLATE_EMULATE, and SPE_MAP_PS.

```

185 {
186     char pathname[256];
187     int i, aff_spe_fd = 0;
188     unsigned int spu_createflags = 0;
189     struct spe_context *spe = NULL;
190     struct spe_context_base_priv *priv;
191
192     /* We need a loader present to run in emulated isolated mode */
193     if (flags & SPE_ISOLATE_EMULATE
194         && !base_spe_emulated_loader_present()) {
195         errno = EINVAL;
196         return NULL;
197     }
198
199     /* Put some sane defaults into the SPE context */
200     spe = malloc(sizeof(*spe));
201     if (!spe) {
202         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
203         return NULL;
204     }
205     memset(spe, 0, sizeof(*spe));
206
207     spe->base_private = malloc(sizeof(*spe->base_private));
208     if (!spe->base_private) {
209         DEBUG_PRINTF("ERROR: Could not allocate "
210                     "spe->base_private context.\n");
211         free(spe);
212         return NULL;
213     }
214
215     /* just a convenience variable */
216     priv = spe->base_private;
217
218     priv->fd_spe_dir = -1;
219     priv->mem_mmap_base = MAP_FAILED;
220     priv->psmap_mmap_base = MAP_FAILED;
221     priv->mssync_mmap_base = MAP_FAILED;
222     priv->mfc_mmap_base = MAP_FAILED;
223     priv->cntl_mmap_base = MAP_FAILED;
224     priv->signal1_mmap_base = MAP_FAILED;
225     priv->signal2_mmap_base = MAP_FAILED;
226     priv->loaded_program = NULL;
227
228     for (i = 0; i < NUM_MBOX_FDS; i++) {
229         priv->spe_fds_array[i] = -1;
230         pthread_mutex_init(&priv->fd_lock[i], NULL);
231     }
232
233     /* initialise spu_createflags */
234     if (flags & SPE_ISOLATE) {
235         flags |= SPE_MAP_PS;
236         spu_createflags |= SPU_CREATE_ISOLATE | SPU_CREATE_NOSCHED;
237     }
238
239     if (flags & SPE_EVENTS_ENABLE)
240         spu_createflags |= SPU_CREATE_EVENTS_ENABLED;
241
242     if (aff_spe)
243         spu_createflags |= SPU_CREATE_AFFINITY_SPU;
244
245     if (flags & SPE_AFFINITY_MEMORY)
246         spu_createflags |= SPU_CREATE_AFFINITY_MEM;
247
248     /* Make the SPUFS directory for the SPE */
249     if (gctx == NULL)
250         sprintf(pathname, "/spu/spthread-%i-%lu",
251                 getpid(), (unsigned long)spe);
252     else
253         sprintf(pathname, "/spu/%s/spthread-%i-%lu",
254                 gctx->base_private->gangname, getpid(),
255                 (unsigned long)spe);
256
257     if (aff_spe)
258         aff_spe_fd = aff_spe->base_private->fd_spe_dir;
259
260     priv->fd_spe_dir = spu_create(pathname, spu_createflags,
261                                 S_IRUSR | S_IWUSR | S_IXUSR, aff_spe_fd);
262
263     if (priv->fd_spe_dir < 0) {
264         int errno_saved = errno; /* save errno to prevent being overwritten */

```

```

265         DEBUG.PRINTF("ERROR: Could not create SPE %s\n", pathname);
266         perror("spu.create()");
267         free_spe_context(spe);
268         /* we mask most errors, but leave ENODEV, etc */
269         switch (errno_saved) {
270             case ENOTSUP:
271             case EEXIST:
272             case EINVAL:
273             case EBUSY:
274             case EPERM:
275             case ENODEV:
276                 errno = errno_saved; /* restore errno */
277                 break;
278             default:
279                 errno = EFAULT;
280                 break;
281         }
282         return NULL;
283     }
284
285     priv->flags = flags;
286
287     /* Map the required areas into process memory */
288     priv->mem_mmap_base = mapfileat(priv->fd_spe_dir, "mem",
LS.SIZE);
289     if (priv->mem_mmap_base == MAP_FAILED) {
290         DEBUG.PRINTF("ERROR: Could not map SPE memory.\n");
291         free_spe_context(spe);
292         errno = ENOMEM;
293         return NULL;
294     }
295
296     if (flags & SPE_MAP_PS) {
297         /* It's possible to map the entire problem state area with
298          * one mmap - try this first */
299         priv->psmap_mmap_base = mapfileat(priv->
fd_spe_dir,
300                                         "psmap", PSMAP_SIZE);
301
302         if (priv->psmap_mmap_base != MAP_FAILED) {
303             priv->mssync_mmap_base =
304                 priv->psmap_mmap_base +
MSSYNC-OFFSET;
305             priv->mfc_mmap_base =
306                 priv->psmap_mmap_base +
MFC-OFFSET;
307             priv->cntl_mmap_base =
308                 priv->psmap_mmap_base +
CNTL-OFFSET;
309             priv->signal1_mmap_base =
310                 priv->psmap_mmap_base +
SIGNAL1-OFFSET;
311             priv->signal2_mmap_base =
312                 priv->psmap_mmap_base +
SIGNAL2-OFFSET;
313         } else {
314             /* map each region separately */
315             priv->mfc_mmap_base =
316                 mapfileat(priv->fd_spe_dir, "mfc",
MFC.SIZE);
317             priv->mssync_mmap_base =
318                 mapfileat(priv->fd_spe_dir, "mss",
MSS.SIZE);
319             priv->cntl_mmap_base =
320                 mapfileat(priv->fd_spe_dir, "cntl",
CNTL.SIZE);
321             priv->signal1_mmap_base =
322                 mapfileat(priv->fd_spe_dir, "signal1",
SIGNAL.SIZE);
323             priv->signal2_mmap_base =
324                 mapfileat(priv->fd_spe_dir, "signal2",
SIGNAL.SIZE);
325
326             if (priv->mfc_mmap_base == MAP_FAILED ||
327                 priv->cntl_mmap_base == MAP_FAILED ||
328                 priv->signal1_mmap_base == MAP_FAILED ||
329                 priv->signal2_mmap_base == MAP_FAILED) {
330                 DEBUG.PRINTF("ERROR: Could not map SPE "
331                             "PS memory.\n");
332             }
333         }
334     }

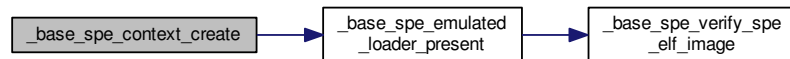
```

```

335         free_spe_context(spe);
336         errno = ENOMEM;
337         return NULL;
338     }
339 }
340 }
341
342 if (flags & SPE_CFG.SIGNOTIFY1_OR) {
343     if (setsignotify(priv->fd.spe_dir, "signal1.type")) {
344         DEBUG_PRINTF("ERROR: Could not open SPE "
345                     "signal1.type file.\n");
346         free_spe_context(spe);
347         errno = EFAULT;
348         return NULL;
349     }
350 }
351
352 if (flags & SPE_CFG.SIGNOTIFY2_OR) {
353     if (setsignotify(priv->fd.spe_dir, "signal2.type")) {
354         DEBUG_PRINTF("ERROR: Could not open SPE "
355                     "signal2.type file.\n");
356         free_spe_context(spe);
357         errno = EFAULT;
358         return NULL;
359     }
360 }
361
362 return spe;
363 }

```

Here is the call graph for this function:



int _base_spe_context_destroy (spe_context_ptr_t *spectx*)

_base_spe_context_destroy cleans up what is left when an SPE executable has exited. Closes open file handles and unmaps memory areas.

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

Definition at line 418 of file create.c.

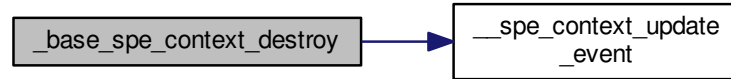
References `__spe_context_update_event()`.

```

419 {
420     int ret = free_spe_context(spe);
421     __spe_context_update_event();
422     return ret;
423 }

```


Here is the call graph for this function:



void _base_spe_context_lock (spe_context_ptr_t spe, enum fd_name fd)

_base_spe_context_lock locks members of the SPE context

Parameters

<i>spectx</i>	Specifies the SPE context
<i>fd</i>	Specifies the file

Definition at line 91 of file create.c.

References spe_context::base_private, and spe_context_base_priv::fd_lock.

Referenced by _base_spe_close_if_open(), and _base_spe_open_if_closed().

```

92 {
93     pthread_mutex_lock(&spe->base_private->fd_lock[fdesc]);
94 }
```

int _base_spe_context_run (spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp, void * envp, spe_stop_info_t * stopinfo)

_base_spe_context_run starts execution of an SPE context with a loaded image

Parameters

<i>spectx</i>	Specifies the SPE context
<i>entry</i>	entry point for the SPE program. If set to 0, entry point is determined by the ELF loader.
<i>runflags</i>	valid values are: SPE_RUN_USER_REGS Specifies that the SPE setup registers r3, r4, and r5 are initialized with the 48 bytes pointed to by argp. SPE_NO_CALLBACKS do not use built in library functions.
<i>argp</i>	An (optional) pointer to application specific data, and is passed as the second parameter to the SPE program.
<i>envp</i>	An (optional) pointer to environment specific data, and is passed as the third parameter to the SPE program.

Definition at line 99 of file run.c.

References _spe_current_active_context, _base_spe_handle_library_callback(), _base_spe_program_load_complete(), spe_context::base_private, DEBUG_PRINTF, spe_context_base_priv::emulated_entry, spe_context_base_priv::entry, spe_context_base_priv::fd_spe_dir, spe_context_base_priv::flags, LS_SIZE, spe_context_base_priv::mem_mmap_base, spe_context_info::npc, spe_context_info::prev, spe_stop_info::result, spe_stop_info::spe_callback_error, SPE_CALLBACK_ERROR, SPE_DEFAULT_ENTRY, SPE_EVENTS_ENABLE, SPE_EXIT, spe_stop_info::spe_exit_code, spe_context_info::spe_id, SPE_ISOLATE, SPE_ISOLATE-EMULATE, spe_stop_info::spe_isolation_error, SPE_ISOLATION_ERROR, SPE_NO_CALLBACKS, S-

PE_PROGRAM_ISO_LOAD_COMPLETE, SPE_PROGRAM_ISOLATED_STOP, SPE_PROGRAM_LIBRARY_CALL, SPE_PROGRAM_NORMAL_END, SPE_RUN_USER_REGS, spe_stop_info::spe_runtime_error, SPE_RUNTIME_ERROR, spe_stop_info::spe_runtime_exception, SPE_RUNTIME_EXCEPTION, spe_stop_info::spe_runtime_fatal, SPE_RUNTIME_FATAL, spe_stop_info::spe_signal_code, SPE_SPU_HALT, SPE_SPU_INVALID_CHANNEL, SPE_SPU_INVALID_INSTR, SPE_SPU_STOPPED_BY_STOP, SPE_SPU_WAITING_ON_CHANNEL, SPE_STOP_AND_SIGNAL, spe_stop_info::spu_status, spe_context_info::status, spe_stop_info::stop_reason, addr64::ui, and addr64::ull.

Referenced by `_event_spe_context_run()`.

```

102 {
103     int retval = 0, run_rc;
104     unsigned int run_status, tmp_entry;
105     spe_stop_info_t stopinfo_buf;
106     struct spe_context_info this_context_info __attribute__((cleanup(
cleanup_speinfo)));
107
108     /* If the caller hasn't set a stopinfo buffer, provide a buffer on the
109      * stack instead. */
110     if (!stopinfo)
111         stopinfo = &stopinfo_buf;
112
113
114     /* In emulated isolated mode, the npc will always return as zero.
115      * use our private entry point instead */
116     if (spe->base_private->flags & SPE_ISOLATE_EMULATE)
117         tmp_entry = spe->base_private->emulated_entry;
118
119     else if (*entry == SPE_DEFAULT_ENTRY)
120         tmp_entry = spe->base_private->entry;
121     else
122         tmp_entry = *entry;
123
124     /* If we're starting the SPE binary from its original entry point,
125      * setup the arguments to main() */
126     if (tmp_entry == spe->base_private->entry &&
127         !(spe->base_private->flags &
128           (SPE_ISOLATE | SPE_ISOLATE_EMULATE))) {
129
130         addr64 argp64, envp64, tid64, ls64;
131         unsigned int regs[128][4];
132
133         /* setup parameters */
134         argp64.ull = (uint64_t)(unsigned long)argp;
135         envp64.ull = (uint64_t)(unsigned long)envp;
136         tid64.ull = (uint64_t)(unsigned long)spe;
137
138         /* make sure the register values are 0 */
139         memset(regs, 0, sizeof(regs));
140
141         /* set sensible values for stack_ptr and stack_size */
142         regs[1][0] = (unsigned int) LS_SIZE - 16;          /* stack_ptr */
143         regs[2][0] = 0;                                     /* stack_size ( 0 =
default ) */
144
145         if (runflags & SPE_RUN_USER_REGS) {
146             /* When SPE_USER_REGS is set, argp points to an array
147              * of 3x128b registers to be passed directly to the SPE
148              * program.
149              */
150             memcpy(regs[3], argp, sizeof(unsigned int) * 12);
151         } else {
152             regs[3][0] = tid64.ui[0];
153             regs[3][1] = tid64.ui[1];
154
155             regs[4][0] = argp64.ui[0];
156             regs[4][1] = argp64.ui[1];
157
158             regs[5][0] = envp64.ui[0];
159             regs[5][1] = envp64.ui[1];
160         }
161
162         /* Store the LS base address in R6 */
163         ls64.ull = (uint64_t)(unsigned long)spe->base_private->
mem_mmap_base;
164         regs[6][0] = ls64.ui[0];
165         regs[6][1] = ls64.ui[1];
166

```

```

167         if (set_regs(spe, regs))
168             return -1;
169     }
170
171     /*Leave a trail of breadcrumbs for the debugger to follow */
172     if (!__spe_current_active_context) {
173         __spe_current_active_context = &this.context_info;
174         if (!__spe_current_active_context)
175             return -1;
176         __spe_current_active_context->prev = NULL;
177     } else {
178         struct spe_context_info *newinfo;
179         newinfo = &this.context_info;
180         if (!newinfo)
181             return -1;
182         newinfo->prev = __spe_current_active_context;
183         __spe_current_active_context = newinfo;
184     }
185     /*remember the ls-addr*/
186     __spe_current_active_context->spe_id = spe->
base.private->fd.spe.dir;
187
188 do_run:
189     /*Remember the npc value*/
190     __spe_current_active_context->npc = tmp.entry;
191
192     /* run SPE context */
193     run_rc = spu_run(spe->base.private->fd.spe.dir,
194                     &tmp.entry, &run.status);
195
196     /*Remember the npc value*/
197     __spe_current_active_context->npc = tmp.entry;
198     __spe_current_active_context->status = run.status;
199
200     DEBUG_PRINTF("spu_run returned run_rc=0x%08x, entry=0x%04x, "
201                 "ext_status=0x%04x.\n", run_rc, tmp.entry, run.status);
202
203     /* set up return values and stopinfo according to spu_run exit
204      * conditions. This is overwritten on error.
205      */
206     stopinfo->spu.status = run_rc;
207
208     if (spe->base.private->flags & SPE_ISOLATE_EMULATE) {
209         /* save the entry point, and pretend that the npc is zero */
210         spe->base.private->emulated_entry = tmp.entry;
211         *entry = 0;
212     } else {
213         *entry = tmp.entry;
214     }
215
216     /* Return with stopinfo set on syscall error paths */
217     if (run_rc == -1) {
218         DEBUG_PRINTF("spu_run returned error %d, errno=%d\n",
219                     run_rc, errno);
220         stopinfo->stop_reason = SPE_RUNTIME_FATAL;
221         stopinfo->result.spe.runtime_fatal = errno;
222         retval = -1;
223
224         /* For isolated contexts, pass EPERM up to the
225          * caller.
226          */
227         if (!(spe->base.private->flags & SPE_ISOLATE
228             && errno == EPERM))
229             errno = EFAULT;
230
231     } else if (run_rc & SPE_SPU_INVALID_INSTR) {
232         DEBUG_PRINTF("SPU has tried to execute an invalid "
233                     "instruction. %d\n", run_rc);
234         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
235         stopinfo->result.spe.runtime_error =
SPE_SPU_INVALID_INSTR;
236         errno = EFAULT;
237         retval = -1;
238
239     } else if ((spe->base.private->flags &
SPE_EVENTS_ENABLE) && run.status) {
240         /* Report asynchronous error if return val are set and
241          * SPU events are enabled.
242          */
243         stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;

```

```

244     stopinfo->result.spe.runtime_exception = run.status;
245     stopinfo->spu.status = -1;
246     errno = EIO;
247     retval = -1;
248
249     } else if (run.rc & SPE_SPU_STOPPED_BY_STOP) {
250         /* Stop & signals are broken down into three groups
251          * 1. SPE library call
252          * 2. SPE user defined stop & signal
253          * 3. SPE program end.
254          *
255          * These groups are signified by the 14-bit stop code:
256          */
257         int stopcode = (run.rc >> 16) & 0x3fff;
258
259         /* Check if this is a library callback, and callbacks are
260          * allowed (ie, running without SPE.NO_CALLBACKS)
261          */
262         if ((stopcode & 0xff00) == SPE_PROGRAM_LIBRARY_CALL
263             && !(runflags & SPE.NO_CALLBACKS)) {
264
265             int callback_rc, callback_number = stopcode & 0xff;
266
267             /* execute library callback */
268             DEBUG_PRINTF("SPE library call: %d\n", callback_number);
269             callback_rc = _base.spe.handle.library_callback(
270
271                                                         callback_number, *entry);
272
273             if (callback_rc) {
274                 /* library callback failed; set errno and
275                  * return immediately */
276                 DEBUG_PRINTF("SPE library call failed: %d\n",
277                             callback_rc);
278                 stopinfo->stop_reason =
279                     SPE_CALLBACK_ERROR;
280
281                 stopinfo->result.spe_callback_error =
282                     callback_rc;
283                 errno = EFAULT;
284                 retval = -1;
285             } else {
286                 /* successful library callback - restart the SPE
287                  * program at the next instruction */
288                 tmp.entry += 4;
289                 goto do.run;
290             }
291
292         } else if ((stopcode & 0xff00) == SPE_PROGRAM_NORMAL_END) {
293             /* The SPE program has exited by exit(X) */
294             stopinfo->stop_reason = SPE_EXIT;
295             stopinfo->result.spe.exit_code = stopcode & 0xff;
296
297             if (spe->base.private->flags &
298                 SPE_ISOLATE) {
299
300                 /* Issue an isolated exit, and re-run the SPE.
301                  * We should see a return value without the
302                  * 0x80 bit set. */
303                 if (!issue.isolated.exit(spe))
304                     goto do.run;
305                 retval = -1;
306             }
307
308         } else if ((stopcode & 0xffff) == SPE_PROGRAM_ISOLATED_STOP) {
309
310             /* 0x2206: isolated app has been loaded by loader;
311              * provide a hook for the debugger to catch this,
312              * and restart
313              */
314             if (stopcode == SPE_PROGRAM_ISO_LOAD_COMPLETE) {
315                 _base.spe.program.load.complete(spe);
316                 goto do.run;
317             } else {
318                 stopinfo->stop_reason =
319                     SPE_ISOLATION_ERROR;
320
321                 stopinfo->result.spe_isolation_error =
322                     stopcode & 0xf;
323             }
324
325         } else if (spe->base.private->flags &
326                     SPE_ISOLATE &&

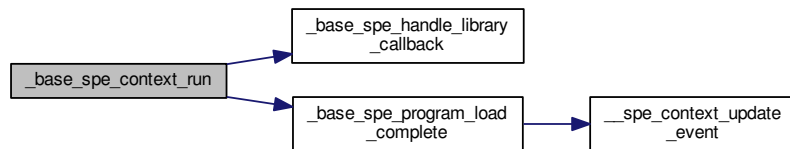
```

```

319             !(run_rc & 0x80)) {
320                 /* We've successfully exited isolated mode */
321                 retval = 0;
322             }
323         } else {
324             /* User defined stop & signal, including
325              * callbacks when disabled */
326             stopinfo->stop_reason = SPE_STOP_AND_SIGNAL;
327             stopinfo->result.spe_signal_code = stopcode;
328             retval = stopcode;
329         }
330     }
331     } else if (run_rc & SPE_SPU_HALT) {
332         DEBUG_PRINTF("SPU was stopped by halt. %d\n", run_rc);
333         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
334         stopinfo->result.spe_runtime_error =
SPE_SPU_HALT;
335         errno = EFAULT;
336         retval = -1;
337     }
338     } else if (run_rc & SPE_SPU_WAITING_ON_CHANNEL) {
339         DEBUG_PRINTF("SPU is waiting on channel. %d\n", run_rc);
340         stopinfo->stop_reason = SPE_RUNTIME_EXCEPTION;
341         stopinfo->result.spe_runtime_exception = run_status;
342         stopinfo->spu_status = -1;
343         errno = EIO;
344         retval = -1;
345     }
346     } else if (run_rc & SPE_SPU_INVALID_CHANNEL) {
347         DEBUG_PRINTF("SPU has tried to access an invalid "
348                     "channel. %d\n", run_rc);
349         stopinfo->stop_reason = SPE_RUNTIME_ERROR;
350         stopinfo->result.spe_runtime_error =
SPE_SPU_INVALID_CHANNEL;
351         errno = EFAULT;
352         retval = -1;
353     }
354     } else {
355         DEBUG_PRINTF("spu.run returned invalid data: 0x%04x\n", run_rc);
356         stopinfo->stop_reason = SPE_RUNTIME_FATAL;
357         stopinfo->result.spe_runtime_fatal = -1;
358         stopinfo->spu_status = -1;
359         errno = EFAULT;
360         retval = -1;
361     }
362 }
363
364 freespeinfo();
365 return retval;
366 }

```

Here is the call graph for this function:



void _base_spe_context_unlock (spe_context_ptr_t spe, enum fd_name fd)

`_base_spe_context_unlock` unlocks members of the SPE context

Parameters

<i>spectx</i>	Specifies the SPE context
<i>fd</i>	Specifies the file

Definition at line 96 of file create.c.

References `spe_context::base_private`, and `spe_context.base_priv::fd.lock`.

Referenced by `_base_spe_close_if_open()`, and `_base_spe_open_if_closed()`.

```

97 {
98     pthread_mutex_unlock(&spe->base_private->fd.lock[fdesc]);
99 }
```

int _base_spe_cpu_info_get (int info_requested, int cpu_node)

`_base_spe_info_get`

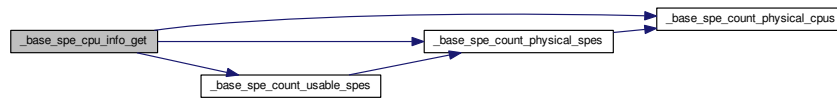
Definition at line 105 of file info.c.

References `_base_spe_count_physical_cpus()`, `_base_spe_count_physical_spes()`, `_base_spe_count_usable_spes()`, `SPE_COUNT_PHYSICAL_CPU_NODES`, `SPE_COUNT_PHYSICAL_SPES`, and `SPE_COUNT_USABLE_SPES`.

```

105                                     {
106     int ret = 0;
107     errno = 0;
108
109     switch (info_requested) {
110     case SPE_COUNT_PHYSICAL_CPU_NODES:
111         ret = _base_spe_count_physical_cpus(cpu_node);
112         break;
113     case SPE_COUNT_PHYSICAL_SPES:
114         ret = _base_spe_count_physical_spes(cpu_node);
115         break;
116     case SPE_COUNT_USABLE_SPES:
117         ret = _base_spe_count_usable_spes(cpu_node);
118         break;
119     default:
120         errno = EINVAL;
121         ret = -1;
122     }
123     return ret;
124 }
```

Here is the call graph for this function:



int _base_spe_emulated_loader_present (void)

Check if the emulated loader is present in the filesystem

Returns

Non-zero if the loader is available, otherwise zero.

Definition at line 159 of file load.c.

References `_base_spe_verify_spe_elf_image()`.

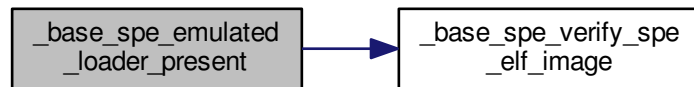
Referenced by `_base_spe_context_create()`.

```

160 {
161     spe_program_handle_t *loader = emulated_loader_program();
162
163     if (!loader)
164         return 0;
165
166     return !_base_spe_verify_spe_elf_image(loader);
167 }

```

Here is the call graph for this function:



spe_gang_context_ptr_t _base_spe_gang_context_create (unsigned int flags)

creates the directory in SPUFS that will contain all SPEs that are considered a gang Note: I would like to generalize this to a "group" or "set" Additional attributes maintained at the group level should be used to define scheduling constraints such "temporal" (e.g., scheduled all at the same time, i.e., a gang) "topology" (e.g., "closeness" of SPEs for optimal communication)

Definition at line 376 of file create.c.

References spe_gang_context::base_private, DEBUG_PRINTF, and spe_gang_context_base_priv::gangname.

```

377 {
378     char pathname[256];
379     struct spe_gang_context_base_priv *pgctx = NULL;
380     struct spe_gang_context *gctx = NULL;
381
382     gctx = malloc(sizeof(*gctx));
383     if (!gctx) {
384         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
385         return NULL;
386     }
387     memset(gctx, 0, sizeof(*gctx));
388
389     pgctx = malloc(sizeof(*pgctx));
390     if (!pgctx) {
391         DEBUG_PRINTF("ERROR: Could not allocate spe context.\n");
392         free(gctx);
393         return NULL;
394     }
395     memset(pgctx, 0, sizeof(*pgctx));
396
397     gctx->base_private = pgctx;
398
399     sprintf(gctx->base_private->gangname, "gang-%i-%lu", getpid(),
400             (unsigned long)gctx);
401     sprintf(pathname, "/spu/%s", gctx->base_private->gangname);
402
403     gctx->base_private->fd.gang_dir = spu_create(pathname, SPU_CREATE_GANG,
404         S_IRUSR | S_IWUSR | S_IXUSR);
405
406     if (gctx->base_private->fd.gang_dir < 0) {
407         DEBUG_PRINTF("ERROR: Could not create Gang %s\n", pathname);
408         free_spe_gang_context(gctx);
409         errno = EFAULT;
410         return NULL;
411     }
412
413     gctx->base_private->flags = flags;

```

```

414
415         return gctx;
416 }

```

int _base_spe_gang_context_destroy (spe_gang_context_ptr_t gctx)

_base_spe_gang_context_destroy destroys a gang context and frees associated resources

Parameters

<i>gctx</i>	Specifies the SPE gang context
-------------	--------------------------------

Definition at line 427 of file create.c.

```

428 {
429     return free_spe_gang_context(gctx);
430 }

```

int _base_spe_image_close (spe_program_handle_t * handle)

_base_spe_image_close unmaps an SPE ELF object that was previously mapped using spe_open_image.

Parameters

<i>handle</i>	handle to open file
---------------	---------------------

Return values

0	On success, spe_close_image returns 0.
-1	On failure, -1 is returned and errno is set appropriately. Possible values for errno: EINVAL From spe_close_image, this indicates that the file, specified by filename, was not previously mapped by a call to spe_open_image.

Definition at line 96 of file image.c.

References spe_program_handle::elf_image, image_handle::map_size, image_handle::speh, and spe_program_handle::toe_shadow.

```

97 {
98     int ret = 0;
99     struct image_handle *ih;
100
101     if (!handle) {
102         errno = EINVAL;
103         return -1;
104     }
105
106     ih = (struct image_handle *)handle;
107
108     if (!ih->speh.elf_image || !ih->map_size) {
109         errno = EINVAL;
110         return -1;
111     }
112
113     if (ih->speh.toe_shadow)
114         free(ih->speh.toe_shadow);
115
116     ret = munmap(ih->speh.elf_image, ih->map_size);
117     free(handle);
118
119     return ret;
120 }

```

spe_program_handle_t* _base_spe_image_open (const char * filename)

_base_spe_image_open maps an SPE ELF executable indicated by filename into system memory and returns the mapped address appropriate for use by the spe_create_thread API. It is often more convenient/appropriate

to use the loading methodologies where SPE ELF objects are converted to PPE static or shared libraries with symbols which point to the SPE ELF objects after these special libraries are loaded. These libraries are then linked with the associated PPE code to provide a direct symbol reference to the SPE ELF object. The symbols in this scheme are equivalent to the address returned from the `spe_open_image` function. SPE ELF objects loaded using this function are not shared with other processes, but SPE ELF objects loaded using the other scheme, mentioned above, can be shared if so desired.

Parameters

<i>filename</i>	Specifies the filename of an SPE ELF executable to be loaded and mapped into system memory.
-----------------	---

Returns

On success, `spe_open_image` returns the address at which the specified SPE ELF object has been mapped. On failure, NULL is returned and `errno` is set appropriately.

Possible values for `errno` include:

EACCES The calling process does not have permission to access the specified file.

EFAULT The filename parameter points to an address that was not contained in the calling process's address space.

A number of other `errno` values could be returned by the `open(2)`, `fstat(2)`, `mmap(2)`, `munmap(2)`, or `close(2)` system calls which may be utilized by the `spe_open_image` or `spe_close_image` functions.

See Also

`spe_create_thread`

Definition at line 37 of file `image.c`.

References `_base_spe_toe_ea()`, `_base_spe_verify_spe_elf_image()`, `spe_program_handle::elf_image`, `spe_program_handle::handle_size`, `image_handle::map_size`, `image_handle::speh`, and `spe_program_handle::toe_shadow`.

```

38 {
39     /* allocate an extra integer in the spe handle to keep the mapped size information */
40     struct image_handle *ret;
41     int binfd = -1, f_stat;
42     struct stat statbuf;
43     size_t ps = getpagesize ();
44
45     ret = malloc(sizeof(struct image_handle));
46     if (!ret)
47         return NULL;
48
49     ret->speh.handle_size = sizeof(spe_program_handle_t);
50     ret->speh.toe_shadow = NULL;
51
52     binfd = open(filename, O_RDONLY);
53     if (binfd < 0)
54         goto ret_err;
55
56     f_stat = fstat(binfd, &statbuf);
57     if (f_stat < 0)
58         goto ret_err;
59
60     /* Sanity: is it executable ?
61     */
62     if (!(statbuf.st_mode & (S_IXUSR | S_IXGRP | S_IXOTH))) {
63         errno=EACCES;
64         goto ret_err;
65     }
66
67     /* now store the size at the extra allocated space */
68     ret->map_size = (statbuf.st_size + ps - 1) & ~(ps - 1);
69
70     ret->speh.elf_image = mmap(NULL, ret->map_size,
71                               PROT_WRITE | PROT_READ,
72                               MAP_PRIVATE, binfd, 0);
73     if (ret->speh.elf_image == MAP_FAILED)

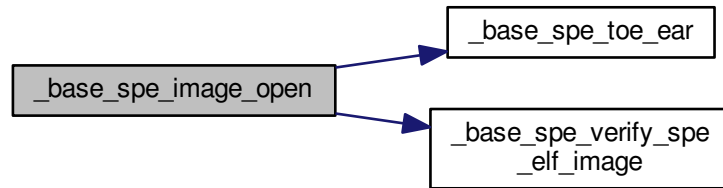
```

```

74         goto ret_err;
75
76         /*Verify that this is a valid SPE ELF object*/
77         if ((_base_spe_verify_spe_elf_image((
spe_program_handle_t *)ret)))
78             goto ret_err;
79
80         if (_base_spe_toe_ear(&ret->speh))
81             goto ret_err;
82
83         /* ok */
84         close(binfd);
85         return (spe_program_handle_t *)ret;
86
87         /* err & cleanup */
88 ret_err:
89         if (binfd >= 0)
90             close(binfd);
91
92         free(ret);
93         return NULL;
94 }

```

Here is the call graph for this function:



int _base_spe_in_mbox_status (spe_context_ptr_t spectx)

The `_base_spe_in_mbox_status` function fetches the status of the SPU inbound mailbox for the SPE thread specified by the `speid` parameter. A 0 value is return if the mailbox is full. A non-zero value specifies the number of available (32-bit) mailbox entries.

Parameters

<i>spectx</i>	Specifies the SPE context whose mailbox status is to be read.
---------------	---

Returns

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

See Also

`spe_read_out_mbox`, `spe_write_in_mbox`, `read` (2)

Definition at line 202 of file `mbox.c`.

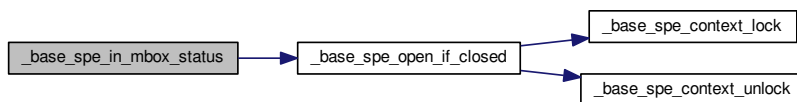
References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `FD_WBOX_STAT`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, and `spe_spu_control_area::SPU_Mbox_Stat`.

```

203 {
204     int rc, ret;
205     volatile struct spe_spu_control_area *cntl_area =
206         spectx->base_private->cntl_mmap_base;
207
208     if (spectx->base_private->flags & SPE_MAP_PS) {
209         ret = (cntl_area->SPU_Mbox_Stat >> 8) & 0xFF;
210     } else {
211         rc = read(_base_spe_open_if_closed(spectx,
212         FD_WBOX_STAT, 0), &ret, 4);
213         if (rc != 4)
214             ret = -1;
215     }
216     return ret;
217 }
218 }

```

Here is the call graph for this function:



int _base_spe_in_mbox_write (spe_context_ptr_t spectx, unsigned int mbox_data[], int count, int behavior_flag)

The _base_spe_in_mbox_write function writes mbox_data to the SPE inbound mailbox for the SPE thread speid.

If the behavior flag indicates ALL_BLOCKING the call will try to write exactly count mailbox entries and block until the write request is satisfied, i.e., exactly count mailbox entries have been written. If the behavior flag indicates ANY_BLOCKING the call will try to write up to count mailbox entries, and block until the write request is satisfied, i.e., at least 1 mailbox entry has been written. If the behavior flag indicates ANY_NON_BLOCKING the call will not block until the write request is satisfied, but instead write whatever is immediately possible and return the number of mailbox entries written. spe_stat_in_mbox can be called to ensure that data can be written prior to calling the function.

Parameters

<i>spectx</i>	Specifies the SPE thread whose outbound mailbox is to be read.
<i>mbox_data</i>	
<i>count</i>	
<i>behavior_flag</i>	ALL_BLOCKING ANY_BLOCKING ANY_NON_BLOCKING

Return values

≥ 0	the number of 32-bit mailbox messages written
----------	---

-1	error condition and errno is set Possible values for errno: EINVAL spectx is invalid Exxxx what else do we need??
----	--

void* _base_spe_ls_area_get (struct spe_context * spectx)

_base_spe_ls_area_get returns a pointer to the start of the memory mapped local store area

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

int _base_spe_ls_size_get (spe_context_ptr_t spe)

_base_spe_ls_size_get returns the size of the local store area

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

Definition at line 105 of file accessors.c.

References LS_SIZE.

```

106 {
107     return LS_SIZE;
108 }
```

int _base_spe_mfcio_get (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

The _base_spe_mfcio_get function places a get DMA command on the proxy command queue of the SPE thread specified by speid. The get command transfers size bytes of data starting at the effective address specified by ea to the local store address specified by ls. The DMA is identified by the tag id specified by tag and performed according to the transfer class and replacement class specified by tid and rid respectively.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 160 of file dma.c.

References MFC_CMD_GET.

```

167 {
168     return spe_do_mfc_get(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_GET);
169 }
```

int _base_spe_mfcio_getb (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

The _base_spe_mfcio_getb function is identical to _base_spe_mfcio_get except that it places a getb (get with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all sequence commands with the same tag identifier as this command are locally ordered with respect to all previously issued commands with the same tag group and command queue.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 171 of file dma.c.

References MFC_CMD_GETB.

```

178 {
179     return spe_do_mfc_get(spectx, ls, ea, size, tag, rid, rid, MFC_CMD_GETB);
180 }
```

int _base_spe_mfcio_getf (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

The _base_spe_mfcio_getf function is identical to _base_spe_mfcio_get except that it places a getf (get with fence) DMA command on the proxy command queue. The fence form ensure that this command is locally ordered with respect to all previously issued commands with the same tag group and command queue.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 182 of file dma.c.

References MFC_CMD_GETF.

```

189 {
190     return spe_do_mfc_get(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_GETF);
191 }
```

int _base_spe_mfcio_put (spe_context_ptr_t *spectx*, unsigned int *ls*, void * *ea*, unsigned int *size*, unsigned int *tag*, unsigned int *tid*, unsigned int *rid*)

The `_base_spe_mfcio_put` function places a put DMA command on the proxy command queue of the SPE thread specified by `speid`. The put command transfers `size` bytes of data starting at the local store address specified by `ls` to the effective address specified by `ea`. The DMA is identified by the tag id specified by `tag` and performed according transfer class and replacement class specified by `tid` and `rid` respectively.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 126 of file `dma.c`.

References `MFC_CMD_PUT`.

```

133 {
134     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUT);
135 }
```

int _base_spe_mfcio_putb (spe_context_ptr_t *spectx*, unsigned int *ls*, void * *ea*, unsigned int *size*, unsigned int *tag*, unsigned int *tid*, unsigned int *rid*)

The `_base_spe_mfcio_putb` function is identical to `_base_spe_mfcio_put` except that it places a putb (put with barrier) DMA command on the proxy command queue. The barrier form ensures that this command and all sequence commands with the same tag identifier as this command are locally ordered with respect to all previously issued commands with the same tag group and command queue.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 137 of file `dma.c`.

References `MFC_CMD_PUTB`.

```

144 {
145     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUTB);
146 }
```

int _base_spe_mfcio_putf (spe_context_ptr_t spectx, unsigned int ls, void * ea, unsigned int size, unsigned int tag, unsigned int tid, unsigned int rid)

The `_base_spe_mfcio_putf` function is identical to `_base_spe_mfcio_put` except that it places a putf (put with fence) DMA command on the proxy command queue. The fence form ensures that this command is locally ordered with respect to all previously issued commands with the same tag group and command queue.

Parameters

<i>spectx</i>	Specifies the SPE context
<i>ls</i>	Specifies the starting local store destination address.
<i>ea</i>	Specifies the starting effective address source address.
<i>size</i>	Specifies the size, in bytes, to be transferred.
<i>tag</i>	Specifies the tag id used to identify the DMA command.
<i>tid</i>	Specifies the transfer class identifier of the DMA command.
<i>rid</i>	Specifies the replacement class identifier of the DMA command.

Returns

On success, return 0. On failure, -1 is returned.

Definition at line 148 of file `dma.c`.

References `MFC_CMD_PUTF`.

```

155 {
156     return spe_do_mfc_put(spectx, ls, ea, size, tag, tid, rid, MFC_CMD_PUTF);
157 }
```

int _base_spe_mfcio_tag_status_read (spe_context_ptr_t spectx, unsigned int mask, unsigned int behavior, unsigned int * tag_status)

`_base_spe_mfcio_tag_status_read`

No Idea

Definition at line 307 of file `dma.c`.

References `spe_context_base_priv::active_tagmask`, `spe_context::base_private`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, `SPE_TAG_ALL`, `SPE_TAG_ANY`, and `SPE_TAG_IMMEDIATE`.

```

308 {
309     if ( mask != 0 ) {
310         if (!(spectx->base_private->flags & SPE_MAP_PS))
311             mask = 0;
312     } else {
313         if (!(spectx->base_private->flags & SPE_MAP_PS))
314             mask = spectx->base_private->active_tagmask;
315     }
316     if (!tag_status) {
317         errno = EINVAL;
318         return -1;
319     }
320     switch (behavior) {
321     case SPE_TAG_ALL:
322         return spe_mfcio_tag_status_read_all(spectx, mask, tag_status);
323     case SPE_TAG_ANY:
324         return spe_mfcio_tag_status_read_any(spectx, mask, tag_status);
325     case SPE_TAG_IMMEDIATE:
326         return spe_mfcio_tag_status_read_immediate(spectx, mask, tag_status);
327     default:
328         errno = EINVAL;
329         return -1;
330     }
331 }
```

int _base_spe_mssync_start (spe_context_ptr_t spectx)

`_base_spe_mssync_start` starts Multisource Synchronisation

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

Definition at line 335 of file dma.c.

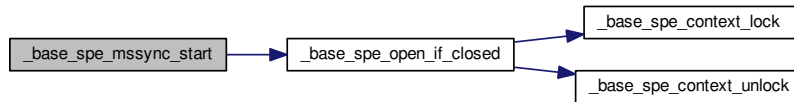
References `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_MSS`, `spe_context_base_priv::flags`, `spe_mssync_area::MFC_MSSync`, `spe_context_base_priv::mssync_mmap_base`, and `SPE_MAP_PS`.

```

336 {
337     int ret, fd;
338     unsigned int data = 1; /* Any value can be written here */
339
340     volatile struct spe_mssync_area *mss_area =
341         spectx->base_private->mssync_mmap_base;
342
343     if (spectx->base_private->flags & SPE_MAP_PS) {
344         mss_area->MFC_MSSync = data;
345         return 0;
346     } else {
347         fd = _base_spe_open_if_closed(spectx,
348             FD_MSS, 0);
349         if (fd != -1) {
350             ret = write(fd, &data, sizeof (data));
351             if ((ret < 0) && (errno != EIO)) {
352                 perror("spe_mssync.start: internal error");
353             }
354             return ret < 0 ? -1 : 0;
355         } else
356             return -1;
357     }
358 }

```

Here is the call graph for this function:



int _base_spe_mssync_status (spe_context_ptr_t spectx)

`_base_spe_mssync_status` retrieves status of Multisource Synchronisation

Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

Definition at line 359 of file dma.c.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_MSS`, `spe_context_base_priv::flags`, `spe_mssync_area::MFC_MSSync`, `spe_context_base_priv::mssync_mmap_base`, and `SPE_MAP_PS`.

```

360 {
361     int ret, fd;
362     unsigned int data;
363
364     volatile struct spe_mssync_area *mss_area =
365         spectx->base_private->mssync_mmap_base;
366
367     if (spectx->base_private->flags & SPE_MAP_PS) {
368         return mss_area->MFC_MSSync;
369     } else {
370         fd = _base_spe_open_if_closed(spectx,
371             FD_MSS, 0);
372         if (fd != -1) {
373             ret = read(fd, &data, sizeof (data));

```

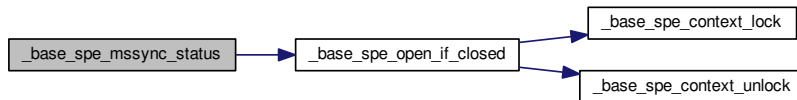


```

373         if ((ret < 0) && (errno != EIO)) {
374             perror("spe_mssync_start: internal error");
375         }
376         return ret < 0 ? -1 : data;
377     } else
378         return -1;
379 }
380 }

```

Here is the call graph for this function:



int _base_spe_out_intr_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[], int count, int behavior_flag)

The `_base_spe_out_intr_mbox_read` function reads the contents of the SPE outbound interrupting mailbox for the SPE context.

Definition at line 255 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `FD_IBOX`, `FD_IBOX_NB`, `SPE_MBOX_ALL_BLOCKING`, `SPE_MBOX_ANY_BLOCKING`, and `SPE_MBOX_ANY_NONBLOCKING`.

```

259 {
260     int rc;
261     int total;
262
263     if (mbox.data == NULL || count < 1){
264         errno = EINVAL;
265         return -1;
266     }
267
268     switch (behavior.flag) {
269     case SPE_MBOX_ALL_BLOCKING: // read all, even if blocking
270         total = rc = 0;
271         while (total < 4*count) {
272             rc = read(_base_spe_open_if_closed(spectx,
273 FD_IBOX, 0),
274                     (char *)mbox.data + total, 4*count - total);
275             if (rc == -1) {
276                 break;
277             }
278             total += rc;
279         }
280         break;
281     case SPE_MBOX_ANY_BLOCKING: // read at least one, even if blocking
282         total = rc = read(_base_spe_open_if_closed(spectx,
283 FD_IBOX, 0), mbox.data, 4*count);
284         break;
285     case SPE_MBOX_ANY_NONBLOCKING: // only read, if non blocking
286         rc = read(_base_spe_open_if_closed(spectx,
287 FD_IBOX_NB, 0), mbox.data, 4*count);
288         if (rc == -1 && errno == EAGAIN) {
289             rc = 0;
290             errno = 0;
291         }
292         total = rc;
293         break;
294     default:
295         errno = EINVAL;

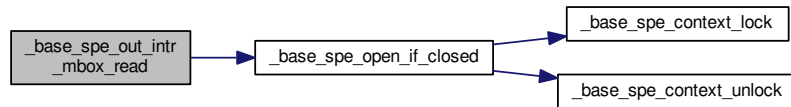
```

```

296         return -1;
297     }
298     if (rc == -1) {
299         errno = EIO;
300         return -1;
301     }
302 }
303
304 return rc / 4;
305 }

```

Here is the call graph for this function:



int _base_spe_out_intr_mbox_status (spe_context_ptr_t spectx)

The `_base_spe_out_intr_mbox_status` function fetches the status of the SPU outbound interrupt mailbox for the SPE thread specified by the `speid` parameter. A 0 value is return if the mailbox is empty. A non-zero value specifies the number of 32-bit unread mailbox entries.

Parameters

<i>spectx</i>	Specifies the SPE context whose mailbox status is to be read.
---------------	---

Returns

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

See Also

`spe_read_out_mbox`, `spe_write_in_mbox`, `read (2)`

Definition at line 238 of file `mbox.c`.

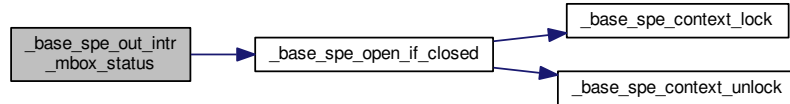
References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `FD_IBOX_STAT`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, and `spe_spu_control_area::SPU_Mbox_Stat`.

```

239 {
240     int rc, ret;
241     volatile struct spe_spu_control_area *cntl_area =
242         spectx->base_private->cntl_mmap_base;
243
244     if (spectx->base_private->flags & SPE_MAP_PS) {
245         ret = (cntl_area->SPU_Mbox_Stat >> 16) & 0xFF;
246     } else {
247         rc = read(_base_spe_open_if_closed(spectx,
248         FD_IBOX_STAT, 0), &ret, 4);
249         if (rc != 4)
250             ret = -1;
251     }
252     return ret;
253 }

```

Here is the call graph for this function:



int _base_spe_out_mbox_read (spe_context_ptr_t spectx, unsigned int mbox_data[], int count)

The _base_spe_out_mbox_read function reads the contents of the SPE outbound interrupting mailbox for the SPE thread speid.

The call will not block until the read request is satisfied, but instead return up to count currently available mailbox entries.

spe_stat_out_intr_mbox can be called to ensure that data is available prior to reading the outbound interrupting mailbox.

Parameters

<i>spectx</i>	Specifies the SPE thread whose outbound mailbox is to be read.
<i>mbox_data</i>	
<i>count</i>	

Return values

>0	the number of 32-bit mailbox messages read
=0	no data available
-1	error condition and errno is set Possible values for errno: EINVAL speid is invalid Exxxx what else do we need??

Definition at line 58 of file mbox.c.

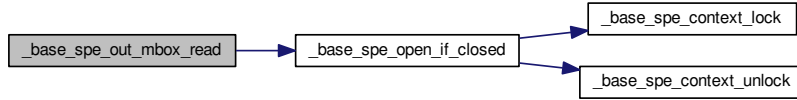
References _base_spe_open_if_closed(), spe_context::base_private, DEBUG_PRINTF, FD_MBOX, spe_context_base_priv::flags, and SPE_MAP_PS.

```

61 {
62     int rc;
63
64     if (mbox_data == NULL || count < 1){
65         errno = EINVAL;
66         return -1;
67     }
68
69     if (spectx->base_private->flags & SPE_MAP_PS) {
70         rc = _base_spe_out_mbox_read_ps(spectx, mbox_data, count);
71     } else {
72         rc = read(_base_spe_open_if_closed(spectx,
73 FD_MBOX, 0), mbox_data, count*4);
74         DEBUG_PRINTF("%s read rc: %d\n", __FUNCTION__, rc);
75         if (rc != -1) {
76             rc /= 4;
77         } else {
78             if (errno == EAGAIN) { // no data ready to be read
79                 errno = 0;
80                 rc = 0;
81             }
82         }
83     }
84     return rc;
85 }

```

Here is the call graph for this function:



int _base_spe_out_mbox_status (spe_context_ptr_t spectx)

The `_base_spe_out_mbox_status` function fetches the status of the SPU outbound mailbox for the SPE thread specified by the `speid` parameter. A 0 value is return if the mailbox is empty. A non-zero value specifies the number of 32-bit unread mailbox entries.

Parameters

<i>spectx</i>	Specifies the SPE context whose mailbox status is to be read.
---------------	---

Returns

On success, returns the current status of the mailbox, respectively. On failure, -1 is returned.

See Also

`spe_read_out_mbox`, `spe_write_in_mbox`, `read` (2)

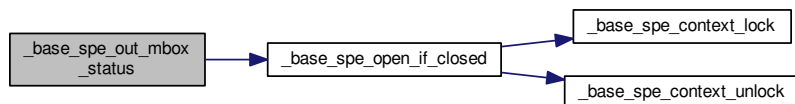
Definition at line 220 of file `mbox.c`.

References `_base_spe_open_if_closed()`, `spe_context::base_private`, `spe_context_base_priv::cntl_mmap_base`, `FD_MBOX_STAT`, `spe_context_base_priv::flags`, `SPE_MAP_PS`, and `spe_spu_control_area::SPU_Mbox_Stat`.

```

221 {
222     int rc, ret;
223     volatile struct spe_spu_control_area *cntl_area =
224         spectx->base_private->cntl_mmap_base;
225
226     if (spectx->base_private->flags & SPE_MAP_PS) {
227         ret = cntl_area->SPU_Mbox_Stat & 0xFF;
228     } else {
229         rc = read(_base_spe_open_if_closed(spectx,
230     FD_MBOX_STAT, 0), &ret, 4);
231         if (rc != 4)
232             ret = -1;
233     }
234     return ret;
235 }
236 }
  
```

Here is the call graph for this function:



int _base_spe_program_load (spe_context_ptr_t *spectx*, spe_program_handle_t * *program*)

_base_spe_program_load loads an ELF image into a context

Parameters

<i>spectx</i>	Specifies the SPE context
<i>program</i>	handle to the ELF image

Definition at line 203 of file load.c.

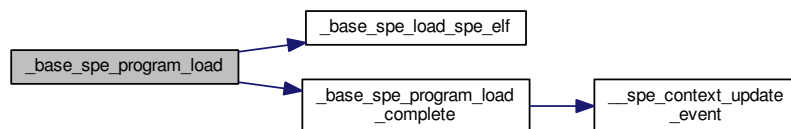
References `_base_spe_load_spe_elf()`, `_base_spe_program_load_complete()`, `spe_context::base_private`, `DEBUG_PRINTF`, `spe_context_base_priv::emulated_entry`, `spe_ld_info::entry`, `spe_context_base_priv::entry`, `spe_context_base_priv::flags`, `spe_context_base_priv::loaded_program`, `spe_context_base_priv::mem_mmap_base`, `SPE_ISOLATE`, and `SPE_ISOLATE_EMULATE`.

```

204 {
205     int rc = 0;
206     struct spe_ld_info ld_info;
207
208     spe->base_private->loaded_program = program;
209
210     if (spe->base_private->flags & SPE_ISOLATE) {
211         rc = spe_start_isolated_app(spe, program);
212     } else if (spe->base_private->flags & SPE_ISOLATE_EMULATE) {
213         rc = spe_start_emulated_isolated_app(spe, program, &ld_info);
214     } else {
215         rc = _base_spe_load_spe_elf(program,
216                                   spe->base_private->mem_mmap_base, &ld_info);
217         if (!rc)
218             _base_spe_program_load_complete(spe);
219     }
220
221     if (rc != 0) {
222         DEBUG_PRINTF ("Load SPE ELF failed..\n");
223         return -1;
224     }
225
226     spe->base_private->entry = ld_info.entry;
227     spe->base_private->emulated_entry = ld_info.entry;
228
229     return 0;
230 }

```

Here is the call graph for this function:



void _base_spe_program_load_complete (spe_context_ptr_t spectx)

Signal that the program load has completed. For normal apps, this is called directly in the load path. For (emulated) isolated apps, the load is asynchronous, so this needs to be called when we know that the load has completed

Precondition

`spe->base_priv->loaded_program` is a valid SPE program

Parameters

<i>spectx</i>	The spe context that has been loaded.
---------------	---------------------------------------

Register the SPE program's start address with the oprofile and gdb, by writing to the object-id file.

Definition at line 38 of file load.c.

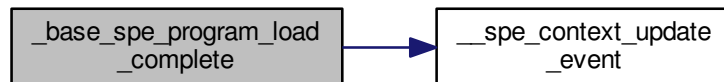
References `__spe_context_update_event()`, `spe_context::base_private`, `DEBUG_PRINTF`, `spe_program._handle::elf_image`, `spe_context_base_priv::fd_spe_dir`, and `spe_context_base_priv::loaded_program`.

Referenced by `_base_spe_context_run()`, and `_base_spe_program_load()`.

```

39 {
40     int objfd, len;
41     char buf[20];
42     spe_program_handle_t *program;
43
44     program = spectx->base_private->loaded_program;
45
46     if (!program || !program->elf_image) {
47         DEBUG_PRINTF("%s called, but no program loaded\n", __func__);
48         return;
49     }
50
51     objfd = openat(spectx->base_private->fd_spe_dir, "object-id", O_RDWR);
52     if (objfd < 0)
53         return;
54
55     len = sprintf(buf, "%p", program->elf_image);
56     write(objfd, buf, len + 1);
57     close(objfd);
58
59     __spe_context_update_event();
60 }
```

Here is the call graph for this function:



void* _base_spe_ps_area_get (struct spe_context * *spectx*, enum ps_area *area*)

_base_spe_ps_area_get returns a pointer to the start of memory mapped problem state area

Parameters

<i>spectx</i>	Specifies the SPE context
<i>area</i>	specifies the area to map

int _base_spe_signal_write (spe_context_ptr_t *spectx*, unsigned int *signal_reg*, unsigned int *data*)

The `_base_spe_signal_write` function writes data to the signal notification register specified by `signal_reg` for the SPE thread specified by the `speid` parameter.

Parameters

<i>spectx</i>	Specifies the SPE context whose signal register is to be written to.
<i>signal_reg</i>	Specified the signal notification register to be written. Valid signal notification registers are: SPE.SIG_NOTIFY_REG.1 SPE signal notification register 1 SPE.SIG_NOTIFY_REG.2 SPE signal notification register 2
<i>data</i>	The 32-bit data to be written to the specified signal notification register.

Returns

On success, `spe_write_signal` returns 0. On failure, -1 is returned.

See Also

`spe_get_ps_area`, `spe_write_in_mbox`

Definition at line 307 of file `mbox.c`.

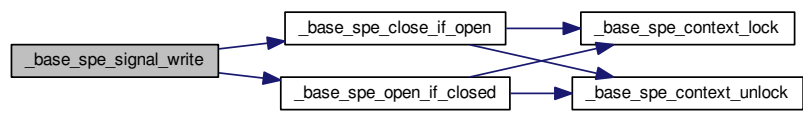
References `_base_spe_close_if_open()`, `_base_spe_open_if_closed()`, `spe_context::base_private`, `FD_SIG1`, `FD_SIG2`, `spe_context_base_priv::flags`, `spe_context_base_priv::signal1_mmap_base`, `spe_context_base_priv::signal2_mmap_base`, `SPE_MAP_PS`, `SPE_SIG_NOTIFY_REG.1`, `SPE_SIG_NOTIFY_REG.2`, `spe_sig_notify_1_area::SPU_Sig_Notify_1`, and `spe_sig_notify_2_area::SPU_Sig_Notify_2`.

```

310 {
311     int rc;
312
313     if (spectx->base_private->flags & SPE_MAP_PS) {
314         if (signal_reg == SPE_SIG_NOTIFY_REG.1) {
315             spe_sig_notify_1_area_t *sig = spectx->
base_private->signal1_mmap_base;
316
317             sig->SPU_Sig_Notify_1 = data;
318         } else if (signal_reg == SPE_SIG_NOTIFY_REG.2) {
319             spe_sig_notify_2_area_t *sig = spectx->
base_private->signal2_mmap_base;
320
321             sig->SPU_Sig_Notify_2 = data;
322         } else {
323             errno = EINVAL;
324             return -1;
325         }
326         rc = 0;
327     } else {
328         if (signal_reg == SPE_SIG_NOTIFY_REG.1)
329             rc = write(_base_spe_open_if_closed(spectx,
FD_SIG1, 0), &data, 4);
330         else if (signal_reg == SPE_SIG_NOTIFY_REG.2)
331             rc = write(_base_spe_open_if_closed(spectx,
FD_SIG2, 0), &data, 4);
332         else {
333             errno = EINVAL;
334             return -1;
335         }
336
337         if (rc == 4)
338             rc = 0;
339
340         if (signal_reg == SPE_SIG_NOTIFY_REG.1)
341             _base_spe_close_if_open(spectx,
FD_SIG1);
342         else if (signal_reg == SPE_SIG_NOTIFY_REG.2)
343             _base_spe_close_if_open(spectx,
FD_SIG2);
344     }
345
346     return rc;
347 }

```


Here is the call graph for this function:



int _base_spe_stop_reason_get (spe_context_ptr_t *spectx*)

_base_spe_stop_reason_get

Parameters

<i>spectx</i>	one thread for which to check why it was stopped
---------------	--

Return values

0	success - eventid and eventdata set appropriately
1	spe has not stopped after checking last, so no data was written to event
-1	an error has happened, event was not touched, errno gets set Possible vales for errno: EINVAL speid is invalid Exxxx what else do we need here??

int _base_spe_stop_status_get (spe_context_ptr_t *spectx*)

_base_spe_stop_status_get

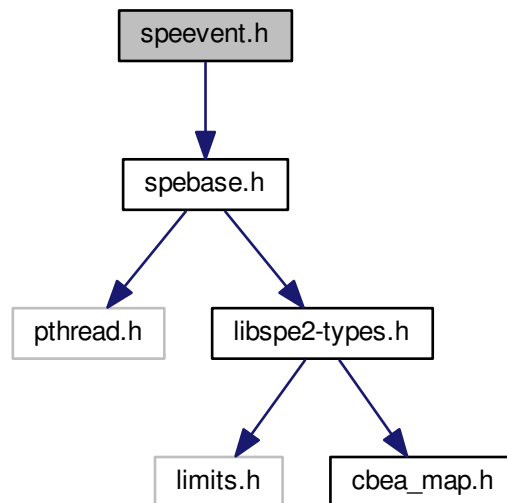
Parameters

<i>spectx</i>	Specifies the SPE context
---------------	---------------------------

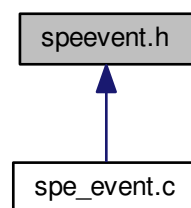
3.24 speevent.h File Reference

```
#include "spebase.h"
```

Include dependency graph for speevent.h:



This graph shows which files directly or indirectly include this file:



Data Structures

- struct `spe_context_event_priv`

Typedefs

- typedef struct
`spe_context_event_priv` `spe_context_event_priv_t`

- typedef struct
spe_context_event_priv * spe_context_event_priv_ptr_t

Enumerations

- enum __spe_event_types {
__SPE_EVENT_OUT_INTR_MBOX, __SPE_EVENT_IN_MBOX, __SPE_EVENT_TAG_GROUP, __SPE_EVENT_SPE_STOPPED,
__NUM_SPE_EVENT_TYPES }

Functions

- int _event_spe_stop_info_read (spe_context_ptr_t spe, spe_stop_info_t *stopinfo)
- spe_event_handler_ptr_t _event_spe_event_handler_create (void)
- int _event_spe_event_handler_destroy (spe_event_handler_ptr_t evhandler)
- int _event_spe_event_handler_register (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)
- int _event_spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t *event)
- int _event_spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t *events, int max_events, int timeout)
- int _event_spe_context_finalize (spe_context_ptr_t spe)
- struct spe_context_event_priv * _event_spe_context_initialize (spe_context_ptr_t spe)
- int _event_spe_context_run (spe_context_ptr_t spe, unsigned int *entry, unsigned int runflags, void *argp, void *envp, spe_stop_info_t *stopinfo)
- void _event_spe_context_lock (spe_context_ptr_t spe)
- void _event_spe_context_unlock (spe_context_ptr_t spe)

3.24.1 Typedef Documentation

typedef struct spe_context_event_priv * spe_context_event_priv_ptr_t

typedef struct spe_context_event_priv spe_context_event_priv_t

3.24.2 Enumeration Type Documentation

enum __spe_event_types

Enumerator

```
__SPE_EVENT_OUT_INTR_MBOX
__SPE_EVENT_IN_MBOX
__SPE_EVENT_TAG_GROUP
__SPE_EVENT_SPE_STOPPED
__NUM_SPE_EVENT_TYPES
```

Definition at line 28 of file speevent.h.

```
28         {
29     __SPE_EVENT_OUT_INTR_MBOX, __SPE_EVENT_IN_MBOX,
30     __SPE_EVENT_TAG_GROUP, __SPE_EVENT_SPE_STOPPED,
31     __NUM_SPE_EVENT_TYPES
32 };
```

3.24.3 Function Documentation

int _event_spe_context_finalize (spe_context_ptr_t spe)

Definition at line 416 of file spe_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_CONTEXT_PRIV_SET`, `spe_context_event_priv::lock`, `spe_context_event_priv::stop_event_pipe`, and `spe_context_event_priv::stop_event_read_lock`.

```

417 {
418     spe_context_event_priv_ptr_t evctx;
419
420     if (!spe) {
421         errno = ESRCH;
422         return -1;
423     }
424
425     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
426     __SPE_EVENT_CONTEXT_PRIV_SET(spe, NULL);
427
428     close(evctx->stop_event_pipe[0]);
429     close(evctx->stop_event_pipe[1]);
430
431     pthread_mutex_destroy(&evctx->lock);
432     pthread_mutex_destroy(&evctx->stop_event_read_lock);
433
434     free(evctx);
435
436     return 0;
437 }
```

struct spe_context_event_priv* _event_spe_context_initialize (spe_context_ptr_t spe)

Definition at line 439 of file spe_event.c.

References `spe_context_event_priv::events`, `spe_context_event_priv::lock`, `spe_event_unit::spe`, `spe_context_event_priv::stop_event_pipe`, and `spe_context_event_priv::stop_event_read_lock`.

```

440 {
441     spe_context_event_priv_ptr_t evctx;
442     int rc;
443     int i;
444
445     evctx = calloc(1, sizeof(*evctx));
446     if (!evctx) {
447         return NULL;
448     }
449
450     rc = pipe(evctx->stop_event_pipe);
451     if (rc == -1) {
452         free(evctx);
453         return NULL;
454     }
455     rc = fcntl(evctx->stop_event_pipe[0], F_GETFL);
456     if (rc != -1) {
457         rc = fcntl(evctx->stop_event_pipe[0], F_SETFL, rc | O_NONBLOCK);
458     }
459     if (rc == -1) {
460         close(evctx->stop_event_pipe[0]);
461         close(evctx->stop_event_pipe[1]);
462         free(evctx);
463         errno = EIO;
464         return NULL;
465     }
466
467     for (i = 0; i < sizeof(evctx->events) / sizeof(evctx->events[0]); i++) {
468         evctx->events[i].spe = spe;
469     }
470
471     pthread_mutex_init(&evctx->lock, NULL);
472     pthread_mutex_init(&evctx->stop_event_read_lock, NULL);
473
474     return evctx;
475 }
```

void _event_spe_context_lock (spe_context_ptr_t spe)

Definition at line 49 of file spe_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`.

Referenced by `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_register()`, and `_event_spe_event_wait()`.

```
50 {
51     pthread_mutex_lock(&__SPE_EVENT_CONTEXT_PRIV_GET(spe)->lock);
52 }
```

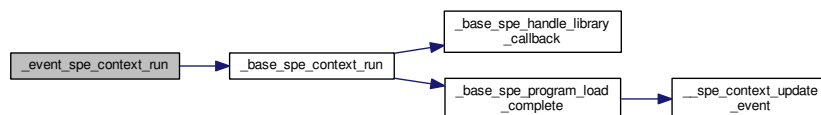
int _event_spe_context_run (spe_context_ptr_t spe, unsigned int * entry, unsigned int runflags, void * argp, void * envp, spe_stop_info_t * stopinfo)

Definition at line 477 of file spe_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`, `_base_spe_context_run()`, and `spe_context_event_priv::stop_event_pipe`.

```
478 {
479     spe_context_event_priv_ptr_t evctx;
480     spe_stop_info_t stopinfo_buf;
481     int rc;
482
483     if (!stopinfo) {
484         stopinfo = &stopinfo_buf;
485     }
486     rc = _base_spe_context_run(spe, entry, runflags, argp, envp, stopinfo);
487
488     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
489     if (write(evctx->stop_event_pipe[1], stopinfo, sizeof(*stopinfo)) != sizeof(*stopinfo)) {
490         /* error check. */
491     }
492
493     return rc;
494 }
```

Here is the call graph for this function:



void _event_spe_context_unlock (spe_context_ptr_t spe)

Definition at line 54 of file spe_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`.

Referenced by `_event_spe_event_handler_deregister()`, `_event_spe_event_handler_register()`, and `_event_spe_event_wait()`.

```
55 {
56     pthread_mutex_unlock(&__SPE_EVENT_CONTEXT_PRIV_GET(spe)->lock);
57 }
```

spe_event_handler_ptr_t _event_spe_event_handler_create (void)

Definition at line 110 of file spe_event.c.

References `__SPE_EPOLL_FD_SET`, and `__SPE_EPOLL_SIZE`.

```

111 {
112     int epfd;
113     spe_event_handler_t *evhandler;
114
115     evhandler = calloc(1, sizeof(*evhandler));
116     if (!evhandler) {
117         return NULL;
118     }
119
120     epfd = epoll_create(__SPE_EPOLL_SIZE);
121     if (epfd == -1) {
122         free(evhandler);
123         return NULL;
124     }
125
126     __SPE_EPOLL_FD_SET(evhandler, epfd);
127
128     return evhandler;
129 }
```

int _event_spe_event_handler_deregister (spe_event_handler_ptr_t evhandler, spe_event_unit_t * event)

Definition at line 273 of file spe_event.c.

References `__base_spe_event_source_acquire()`, `__SPE_EPOLL_FD_GET`, `__SPE_EVENT_ALL`, `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_IN_MBOX`, `__SPE_EVENT_OUT_INTR_MBOX`, `__SPE_EVENT_SPE_STOPPED`, `__SPE_EVENT_TAG_GROUP`, `__SPE_EVENTS_ENABLED`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, `spe_context_event_priv::events`, `spe_event_unit::events`, `FD_IBOX`, `FD_MFC`, `FD_WBOX`, `spe_event_unit::spe`, `SPE_EVENT_IN_MBOX`, `SPE_EVENT_OUT_INTR_MBOX`, `SPE_EVENT_SPE_STOPPED`, `SPE_EVENT_TAG_GROUP`, and `spe_context_event_priv::stop_event_pipe`.

```

274 {
275     int epfd;
276     const int ep_op = EPOLL_CTL_DEL;
277     spe_context_event_priv_ptr_t evctx;
278     int fd;
279
280     if (!evhandler) {
281         errno = ESRCH;
282         return -1;
283     }
284     if (!event || !event->spe) {
285         errno = EINVAL;
286         return -1;
287     }
288     if (!__SPE_EVENTS_ENABLED(event->spe)) {
289         errno = ENOTSUP;
290         return -1;
291     }
292
293     epfd = __SPE_EPOLL_FD_GET(evhandler);
294     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);
295
296     if (event->events & ~__SPE_EVENT_ALL) {
297         errno = ENOTSUP;
298         return -1;
299     }
300
301     _event_spe_context_lock(event->spe); /* for spe->event.private->events */
302
303     if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
304         fd = __base_spe_event_source_acquire(event->
305         spe, FD_IBOX);
306         if (fd == -1) {
307             _event_spe_context_unlock(event->spe);
308             return -1;
309         }
310         if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {

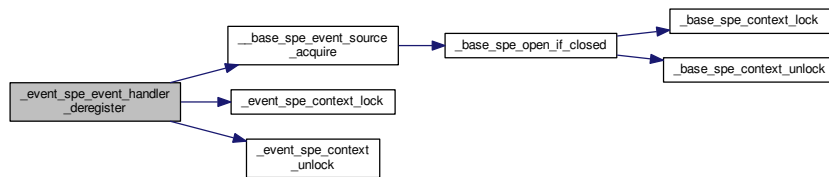
```

```

310     _event_spe_context_unlock(event->spe);
311     return -1;
312 }
313 evctx->events[__SPE_EVENT_OUT_INTR_MBOX].
events = 0;
314 }
315
316 if (event->events & SPE_EVENT_IN_MBOX) {
317     fd = _base_spe_event_source_acquire(event->
spe, FD_WBOX);
318     if (fd == -1) {
319         _event_spe_context_unlock(event->spe);
320         return -1;
321     }
322     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
323         _event_spe_context_unlock(event->spe);
324         return -1;
325     }
326     evctx->events[__SPE_EVENT_IN_MBOX].events = 0;
327 }
328
329 if (event->events & SPE_EVENT_TAG_GROUP) {
330     fd = _base_spe_event_source_acquire(event->
spe, FD_MFC);
331     if (fd == -1) {
332         _event_spe_context_unlock(event->spe);
333         return -1;
334     }
335     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
336         _event_spe_context_unlock(event->spe);
337         return -1;
338     }
339     evctx->events[__SPE_EVENT_TAG_GROUP].events = 0;
340 }
341
342 if (event->events & SPE_EVENT_SPE_STOPPED) {
343     fd = evctx->stop_event_pipe[0];
344     if (epoll_ctl(epfd, ep_op, fd, NULL) == -1) {
345         _event_spe_context_unlock(event->spe);
346         return -1;
347     }
348     evctx->events[__SPE_EVENT_SPE_STOPPED].events = 0;
349 }
350
351 _event_spe_context_unlock(event->spe);
352
353 return 0;
354 }

```

Here is the call graph for this function:



int _event_spe_event_handler_destroy (spe_event_handler_ptr_t evhandler)

Definition at line 135 of file spe_event.c.

References `__SPE_EPOLL_FD_GET`.

```

136 {
137     int epfd;
138
139     if (!evhandler) {

```

```

140     errno = ESRCH;
141     return -1;
142 }
143
144 epfd = __SPE_EPOLL_FD_GET(evhandler);
145 close(epfd);
146
147 free(evhandler);
148 return 0;
149 }

```

int _event_spe_event_handler_register (spe_event_handler_ptr_t evhandler, spe_event_unit_t * event)

Definition at line 155 of file spe_event.c.

References `_base_spe_event_source_acquire()`, `__SPE_EPOLL_FD_GET`, `__SPE_EVENT_ALL`, `__SPE_EVENT_CONTEXT_PRIV_GET`, `__SPE_EVENT_IN_MBOX`, `__SPE_EVENT_OUT_INTR_MBOX`, `__SPE_EVENT_SPE_STOPPED`, `__SPE_EVENT_TAG_GROUP`, `__SPE_EVENTS_ENABLED`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, `spe_context::base_private`, `spe_event_unit::data`, `spe_context_event_priv::events`, `spe_event_unit::events`, `FD_IBOX`, `FD_MFC`, `FD_WBOX`, `spe_context_base_priv::flags`, `spe_event_data::ptr`, `spe_event_unit::spe`, `SPE_EVENT_IN_MBOX`, `SPE_EVENT_OUT_INTR_MBOX`, `SPE_EVENT_SPE_STOPPED`, `SPE_EVENT_TAG_GROUP`, `SPE_MAP_PS`, and `spe_context_event_priv::stop_event_pipe`.

```

156 {
157     int epfd;
158     const int ep_op = EPOLL_CTL_ADD;
159     spe_context_event_priv_ptr_t evctx;
160     spe_event_unit_t *ev_buf;
161     struct epoll_event ep_event;
162     int fd;
163
164     if (!evhandler) {
165         errno = ESRCH;
166         return -1;
167     }
168     if (!event || !event->spe) {
169         errno = EINVAL;
170         return -1;
171     }
172     if (!__SPE_EVENTS_ENABLED(event->spe)) {
173         errno = ENOTSUP;
174         return -1;
175     }
176
177     epfd = __SPE_EPOLL_FD_GET(evhandler);
178     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(event->spe);
179
180     if (event->events & ~__SPE_EVENT_ALL) {
181         errno = ENOTSUP;
182         return -1;
183     }
184
185     _event_spe_context_lock(event->spe); /* for spe->event_private->events */
186
187     if (event->events & SPE_EVENT_OUT_INTR_MBOX) {
188         fd = _base_spe_event_source_acquire(event->
189         spe, FD_IBOX);
189         if (fd == -1) {
190             _event_spe_context_unlock(event->spe);
191             return -1;
192         }
193
194         ev_buf = &evctx->events[__SPE_EVENT_OUT_INTR_MBOX];
195         ev_buf->events = SPE_EVENT_OUT_INTR_MBOX;
196         ev_buf->data = event->data;
197
198         ep_event.events = EPOLLIN;
199         ep_event.data.ptr = ev_buf;
200         if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
201             _event_spe_context_unlock(event->spe);
202             return -1;
203         }
204     }
205 }

```

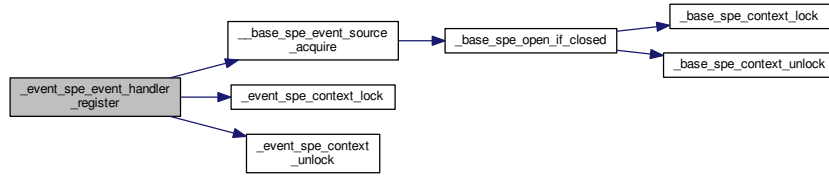


```

205
206     if (event->events & SPE_EVENT_IN_MBOX) {
207         fd = __base_spe_event_source_acquire(event->
spe, FD_WBOX);
208         if (fd == -1) {
209             _event_spe_context_unlock(event->spe);
210             return -1;
211         }
212
213         ev_buf = &evctx->events[__SPE_EVENT_IN_MBOX];
214         ev_buf->events = SPE_EVENT_IN_MBOX;
215         ev_buf->data = event->data;
216
217         ep_event.events = EPOLLOUT;
218         ep_event.data.ptr = ev_buf;
219         if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
220             _event_spe_context_unlock(event->spe);
221             return -1;
222         }
223     }
224
225     if (event->events & SPE_EVENT_TAG_GROUP) {
226         fd = __base_spe_event_source_acquire(event->
spe, FD_MFC);
227         if (fd == -1) {
228             _event_spe_context_unlock(event->spe);
229             return -1;
230         }
231
232         if (event->spe->base_private->flags & SPE_MAP_PS) {
233             _event_spe_context_unlock(event->spe);
234             errno = ENOTSUP;
235             return -1;
236         }
237
238         ev_buf = &evctx->events[__SPE_EVENT_TAG_GROUP];
239         ev_buf->events = SPE_EVENT_TAG_GROUP;
240         ev_buf->data = event->data;
241
242         ep_event.events = EPOLLIN;
243         ep_event.data.ptr = ev_buf;
244         if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
245             _event_spe_context_unlock(event->spe);
246             return -1;
247         }
248     }
249
250     if (event->events & SPE_EVENT_SPE_STOPPED) {
251         fd = evctx->stop_event_pipe[0];
252
253         ev_buf = &evctx->events[__SPE_EVENT_SPE_STOPPED];
254         ev_buf->events = SPE_EVENT_SPE_STOPPED;
255         ev_buf->data = event->data;
256
257         ep_event.events = EPOLLIN;
258         ep_event.data.ptr = ev_buf;
259         if (epoll_ctl(epfd, ep_op, fd, &ep_event) == -1) {
260             _event_spe_context_unlock(event->spe);
261             return -1;
262         }
263     }
264
265     _event_spe_context_unlock(event->spe);
266
267     return 0;
268 }

```

Here is the call graph for this function:



int _event_spe_event_wait (spe_event_handler_ptr_t evhandler, spe_event_unit_t * events, int max_events, int timeout)

Definition at line 360 of file spe_event.c.

References `__SPE_EPOLL_FD_GET`, `_event_spe_context_lock()`, `_event_spe_context_unlock()`, and `spe_event_unit::spe`.

```

361 {
362     int epfd;
363     struct epoll_event *ep_events;
364     int rc;
365
366     if (!evhandler) {
367         errno = ESRCH;
368         return -1;
369     }
370     if (!events || max_events <= 0) {
371         errno = EINVAL;
372         return -1;
373     }
374
375     epfd = __SPE_EPOLL_FD_GET(evhandler);
376
377     ep_events = malloc(sizeof(*ep_events) * max_events);
378     if (!ep_events) {
379         return -1;
380     }
381
382     for ( ; ; ) {
383         rc = epoll_wait(epfd, ep_events, max_events, timeout);
384         if (rc == -1) { /* error */
385             if (errno == EINTR) {
386                 if (timeout >= 0) { /* behave as timeout */
387                     rc = 0;
388                     break;
389                 }
390                 /* else retry */
391             }
392             else {
393                 break;
394             }
395         }
396         else if (rc > 0) {
397             int i;
398             for (i = 0; i < rc; i++) {
399                 spe_event_unit_t *ev = (spe_event_unit_t *) (ep_events[i].data.ptr);
400                 _event_spe_context_lock(ev->spe); /* lock ev itself */
401                 events[i] = *ev;
402                 _event_spe_context_unlock(ev->spe);
403             }
404             break;
405         }
406         else { /* timeout */
407             break;
408         }
409     }
410 }

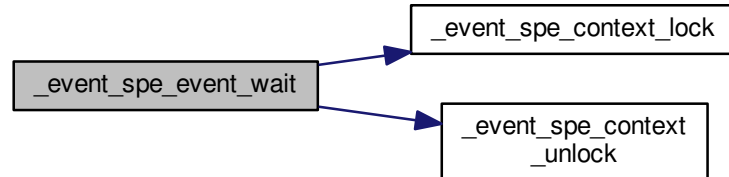
```

```

411  free(ep.events);
412
413  return rc;
414 }

```

Here is the call graph for this function:



int _event_spe_stop_info_read (spe_context_ptr_t spe, spe_stop_info_t * stopinfo)

Definition at line 59 of file spe_event.c.

References `__SPE_EVENT_CONTEXT_PRIV_GET`, `spe_context_event_priv::stop_event_pipe`, and `spe_context_event_priv::stop_event_read_lock`.

```

60 {
61     spe_context_event_priv_ptr_t evctx;
62     int rc;
63     int fd;
64     size_t total;
65
66     evctx = __SPE_EVENT_CONTEXT_PRIV_GET(spe);
67     fd = evctx->stop_event_pipe[0];
68
69     pthread_mutex_lock(&evctx->stop_event_read_lock); /* for atomic read */
70
71     rc = read(fd, stopinfo, sizeof(*stopinfo));
72     if (rc == -1) {
73         pthread_mutex_unlock(&evctx->stop_event_read_lock);
74         return -1;
75     }
76
77     total = rc;
78     while (total < sizeof(*stopinfo)) { /* this loop will be executed in few cases */
79         struct pollfd fds;
80         fds.fd = fd;
81         fds.events = POLLIN;
82         rc = poll(&fds, 1, -1);
83         if (rc == -1) {
84             if (errno != EINTR) {
85                 break;
86             }
87         }
88         else if (rc == 1) {
89             rc = read(fd, (char *)stopinfo + total, sizeof(*stopinfo) - total);
90             if (rc == -1) {
91                 if (errno != EAGAIN) {
92                     break;
93                 }
94             }
95             else {
96                 total += rc;
97             }
98         }
99     }
100 }

```

```
101 pthread_mutex_unlock(&evctx->stop_event_read_lock);
102
103 return rc == -1 ? -1 : 0;
104 }
```

Index

- __NUM_SPE_EVENT_TYPES
 - speevent.h, 159
- __SPE_EVENT_IN_MBOX
 - speevent.h, 159
- __SPE_EVENT_OUT_INTR_MBOX
 - speevent.h, 159
- __SPE_EVENT_SPE_STOPPED
 - speevent.h, 159
- __SPE_EVENT_TAG_GROUP
 - speevent.h, 159
- __PRINTF
 - elf_loader.c, 58
 - handler_utils.h, 67
 - spebase.h, 124
- __SPE_EVENT_ALL
 - spe_event.c, 113
- __attribute__
 - accessors.c, 36
- __base_spe_event_source_acquire
 - accessors.c, 36
 - spebase.h, 126
- __base_spe_event_source_release
 - accessors.c, 36
 - spebase.h, 127
- __base_spe_spe_dir_get
 - accessors.c, 37
 - spebase.h, 127
- __base_spe_stop_event_source_get
 - accessors.c, 37
 - spebase.h, 127
- __base_spe_stop_event_target_get
 - accessors.c, 37
 - spebase.h, 127
- __reserved_ptr
 - spe_stop_info, 33
- __reserved_u64
 - spe_stop_info, 33
- __spe_context_update_event
 - spebase.h, 128
- __spe_current_active_context
 - run.c, 111
- __spe_event_types
 - speevent.h, 159
- __base_spe_callback_handler_deregister
 - lib_builtin.c, 78
- spebase.h, 128
- __base_spe_callback_handler_query
 - lib_builtin.c, 78
 - spebase.h, 128
- __base_spe_callback_handler_register
 - lib_builtin.c, 79
 - spebase.h, 128
- __base_spe_close_if_open
 - create.c, 41
 - create.h, 48
- __base_spe_context_create
 - create.c, 41
 - spebase.h, 129
- __base_spe_context_destroy
 - create.c, 44
 - spebase.h, 132
- __base_spe_context_lock
 - create.c, 45
 - spebase.h, 133
- __base_spe_context_run
 - run.c, 107
 - spebase.h, 133
- __base_spe_context_unlock
 - create.c, 45
 - spebase.h, 137
- __base_spe_count_physical_cpus
 - info.c, 72
 - info.h, 76
- __base_spe_count_physical_spes
 - info.c, 73
 - info.h, 76
- __base_spe_count_usable_spes
 - info.c, 73
 - info.h, 77
- __base_spe_cpu_info_get
 - info.c, 74
 - spebase.h, 138
- __base_spe_emulated_loader_present
 - load.c, 94
 - spebase.h, 138
- __base_spe_gang_context_create
 - create.c, 45
 - spebase.h, 139
- __base_spe_gang_context_destroy
 - create.c, 46

- spebase.h, 140
- _base_spe_handle_library_callback
 - lib_builtin.c, 79
 - lib_builtin.h, 81
- _base_spe_image_close
 - image.c, 69
 - spebase.h, 140
- _base_spe_image_open
 - image.c, 70
 - spebase.h, 140
- _base_spe_in_mbox_status
 - mbox.c, 97
 - spebase.h, 142
- _base_spe_in_mbox_write
 - mbox.c, 98
 - spebase.h, 143
- _base_spe_load_spe_elf
 - elf_loader.c, 58
 - elf_loader.h, 63
- _base_spe_ls_area_get
 - accessors.c, 37
 - spebase.h, 144
- _base_spe_ls_size_get
 - accessors.c, 37
 - spebase.h, 144
- _base_spe_mfcio_get
 - dma.c, 50
 - spebase.h, 144
- _base_spe_mfcio_getb
 - dma.c, 51
 - spebase.h, 144
- _base_spe_mfcio_getf
 - dma.c, 51
 - spebase.h, 145
- _base_spe_mfcio_put
 - dma.c, 52
 - spebase.h, 145
- _base_spe_mfcio_putb
 - dma.c, 52
 - spebase.h, 146
- _base_spe_mfcio_putf
 - dma.c, 53
 - spebase.h, 146
- _base_spe_mfcio_tag_status_read
 - dma.c, 53
 - spebase.h, 147
- _base_spe_mssync_start
 - dma.c, 54
 - spebase.h, 147
- _base_spe_mssync_status
 - dma.c, 55
 - spebase.h, 148
- _base_spe_open_if_closed
 - create.c, 46
- create.h, 48
- _base_spe_out_intr_mbox_read
 - mbox.c, 100
 - spebase.h, 149
- _base_spe_out_intr_mbox_status
 - mbox.c, 101
 - spebase.h, 150
- _base_spe_out_mbox_read
 - mbox.c, 101
 - spebase.h, 151
- _base_spe_out_mbox_status
 - mbox.c, 102
 - spebase.h, 152
- _base_spe_parse_isolated_elf
 - elf_loader.c, 59
 - elf_loader.h, 64
- _base_spe_program_load
 - load.c, 95
 - spebase.h, 152
- _base_spe_program_load_complete
 - load.c, 96
 - spebase.h, 154
- _base_spe_ps_area_get
 - accessors.c, 38
 - spebase.h, 155
- _base_spe_signal_write
 - mbox.c, 103
 - spebase.h, 155
- _base_spe_stop_reason_get
 - spebase.h, 157
- _base_spe_stop_status_get
 - spebase.h, 157
- _base_spe_toe_ear
 - elf_loader.c, 60
 - elf_loader.h, 65
- _base_spe_verify_spe_elf_image
 - elf_loader.c, 61
 - elf_loader.h, 66
- _event_spe_context_finalize
 - spe_event.c, 113
 - speevent.h, 160
- _event_spe_context_initialize
 - spe_event.c, 114
 - speevent.h, 160
- _event_spe_context_lock
 - spe_event.c, 114
 - speevent.h, 160
- _event_spe_context_run
 - spe_event.c, 115
 - speevent.h, 161
- _event_spe_context_unlock
 - spe_event.c, 115
 - speevent.h, 161
- _event_spe_event_handler_create

- spe_event.c, 115
- speevent.h, 161
- _event_spe_event_handler_deregister
 - spe_event.c, 116
 - speevent.h, 162
- _event_spe_event_handler_destroy
 - spe_event.c, 117
 - speevent.h, 163
- _event_spe_event_handler_register
 - spe_event.c, 117
 - speevent.h, 164
- _event_spe_event_wait
 - spe_event.c, 119
 - speevent.h, 166
- _event_spe_stop_info_read
 - spe_event.c, 121
 - speevent.h, 167
- accessors.c, 35
 - _attribute_, 36
 - _base_spe_event_source_acquire, 36
 - _base_spe_event_source_release, 36
 - _base_spe_spe_dir_get, 37
 - _base_spe_stop_event_source_get, 37
 - _base_spe_stop_event_target_get, 37
 - _base_spe_ls_area_get, 37
 - _base_spe_ls_size_get, 37
 - _base_spe_ps_area_get, 38
- active_tagmask
 - spe_context_base_priv, 13
- addr64, 5
 - ui, 5
 - ull, 5
- base_private
 - spe_context, 11
 - spe_gang_context, 22
- CNTL_OFFSET
 - spebase.h, 124
- CNTL_SIZE
 - spebase.h, 124
- cbea_map.h, 39
 - spe_mfc_command_area_t, 40
 - spe_mssync_area_t, 40
 - spe_sig_notify_1_area_t, 40
 - spe_sig_notify_2_area_t, 40
 - spe_spu_control_area_t, 40
- class
 - mfc_command_parameter_area, 8
- cmd
 - mfc_command_parameter_area, 8
- cntl_mmap_base
 - spe_context_base_priv, 13
- create.c, 40
 - _base_spe_close_if_open, 41
 - _base_spe_context_create, 41
 - _base_spe_context_destroy, 44
 - _base_spe_context_lock, 45
 - _base_spe_context_unlock, 45
 - _base_spe_gang_context_create, 45
 - _base_spe_gang_context_destroy, 46
 - _base_spe_open_if_closed, 46
- create.h, 47
 - _base_spe_close_if_open, 48
 - _base_spe_open_if_closed, 48
- DEBUG_PRINTF
 - elf_loader.c, 58
 - handler_utils.h, 67
 - spebase.h, 124
- DECL_0_ARGS
 - handler_utils.h, 67
- DECL_1_ARGS
 - handler_utils.h, 67
- DECL_2_ARGS
 - handler_utils.h, 67
- DECL_3_ARGS
 - handler_utils.h, 67
- DECL_4_ARGS
 - handler_utils.h, 67
- DECL_5_ARGS
 - handler_utils.h, 67
- DECL_6_ARGS
 - handler_utils.h, 67
- DECL_RET
 - handler_utils.h, 68
- data
 - spe_event_unit, 20
- design.txt, 49
- dma.h
 - MFC_CMD_GET, 57
 - MFC_CMD_GETB, 57
 - MFC_CMD_GETF, 57
 - MFC_CMD_PUT, 57
 - MFC_CMD_PUTB, 57
 - MFC_CMD_PUTF, 57
- dma.c, 49
 - _base_spe_mfcio_get, 50
 - _base_spe_mfcio_getb, 51
 - _base_spe_mfcio_getf, 51
 - _base_spe_mfcio_put, 52
 - _base_spe_mfcio_putb, 52
 - _base_spe_mfcio_putf, 53
 - _base_spe_mfcio_tag_status_read, 53
 - _base_spe_mssync_start, 54
 - _base_spe_mssync_status, 55
- dma.h, 55

- mfc_cmd, 57
- ea
 - mfc_command_parameter_area, 8
- elf_image
 - spe_program_handle, 27
- elf_loader.c, 57
 - _PRINTF, 58
 - _base_spe_load_spe_elf, 58
 - _base_spe_parse_isolated_elf, 59
 - _base_spe_toe_ear, 60
 - _base_spe_verify_spe_elf_image, 61
 - DEBUG_PRINTF, 58
 - TAG, 58
- elf_loader.h, 61
 - _base_spe_load_spe_elf, 63
 - _base_spe_parse_isolated_elf, 64
 - _base_spe_toe_ear, 65
 - _base_spe_verify_spe_elf_image, 66
 - LS_SIZE, 63
 - SPE_LDR_PARAMS_start, 63
 - SPE_LDR_PROG_start, 63
- emulated_entry
 - spe_context_base_priv, 13
- entry
 - spe_context_base_priv, 13
 - spe_ld_info, 23
- ev_pipe
 - spe_context_base_priv, 13
- event_private
 - spe_context, 11
 - spe_gang_context, 22
- events
 - spe_context_event_priv, 17
 - spe_event_unit, 21
- FD_IBOX
 - spebase.h, 126
- FD_IBOX_NB
 - spebase.h, 126
- FD_IBOX_STAT
 - spebase.h, 126
- FD_MBOX
 - spebase.h, 126
- FD_MBOX_STAT
 - spebase.h, 126
- FD_MFC
 - spebase.h, 126
- FD_MSS
 - spebase.h, 126
- FD_SIG1
 - spebase.h, 126
- FD_SIG2
 - spebase.h, 126
- FD_WBOX
 - spebase.h, 126
- FD_WBOX_NB
 - spebase.h, 126
- FD_WBOX_STAT
 - spebase.h, 126
- fd_attr, 6
 - mode, 6
 - name, 6
- fd_gang_dir
 - spe_gang_context_base_priv, 23
- fd_grp_dir
 - spe_context_base_priv, 13
- fd_lock
 - spe_context_base_priv, 13
- fd_name
 - spebase.h, 126
- fd_spe_dir
 - spe_context_base_priv, 13
- flags
 - spe_context_base_priv, 13
 - spe_gang_context_base_priv, 23
- GET_LS_PTR
 - handler_utils.h, 68
- GET_LS_PTR_NULL
 - handler_utils.h, 68
- GNU_SOURCE
 - run.c, 107
- gangname
 - spe_gang_context_base_priv, 23
- HANDLER_IDX
 - lib_builtin.c, 78
- handle
 - spe_context, 11
- handle_size
 - spe_program_handle, 27
- handler_utils.h, 66
 - _PRINTF, 67
 - DEBUG_PRINTF, 67
 - DECL_0_ARGS, 67
 - DECL_1_ARGS, 67
 - DECL_2_ARGS, 67
 - DECL_3_ARGS, 67
 - DECL_4_ARGS, 67
 - DECL_5_ARGS, 67
 - DECL_6_ARGS, 67
 - DECL_RET, 68
 - GET_LS_PTR, 68
 - GET_LS_PTR_NULL, 68
 - LS_ADDR_MASK, 68
 - LS_ARG_ADDR, 68
 - LS_SIZE, 68

- PUT_LS_RC, 68
- image.c, 68
 - _base_spe_image_close, 69
 - _base_spe_image_open, 70
- image_handle, 7
 - map_size, 7
 - speh, 7
- info.c, 71
 - _base_spe_count_physical_cpus, 72
 - _base_spe_count_physical_spes, 73
 - _base_spe_count_usable_spes, 73
 - _base_spe_cpu_info_get, 74
- info.h, 74
 - _base_spe_count_physical_cpus, 76
 - _base_spe_count_physical_spes, 76
 - _base_spe_count_usable_spes, 77
 - THREADS_PER_BE, 76
- LS_ADDR_MASK
 - handler_utils.h, 68
- LS_ARG_ADDR
 - handler_utils.h, 68
- LS_SIZE
 - elf_loader.h, 63
 - handler_utils.h, 68
 - spebase.h, 124
- lib_builtin.c, 77
 - _base_spe_callback_handler_deregister, 78
 - _base_spe_callback_handler_query, 78
 - _base_spe_callback_handler_register, 79
 - _base_spe_handle_library_callback, 79
 - HANDLER_IDX, 78
- lib_builtin.h, 80
 - _base_spe_handle_library_callback, 81
 - MAX_CALLNUM, 81
 - RESERVED, 81
- libspe2-types.h
 - SPE_CONTROL_AREA, 89
 - SPE_MFC_COMMAND_AREA, 89
 - SPE_MSSYNC_AREA, 89
 - SPE_SIG_NOTIFY_1_AREA, 89
 - SPE_SIG_NOTIFY_2_AREA, 89
- libspe2-types.h, 82
 - ps_area, 89
 - SIGSPE, 84
 - SPE_AFFINITY_MEMORY, 84
 - SPE_CALLBACK_ERROR, 84
 - SPE_CALLBACK_NEW, 84
 - SPE_CALLBACK_UPDATE, 84
 - SPE_DEFAULT_ENTRY, 85
 - SPE_DMA_ALIGNMENT, 85
 - SPE_DMA_STORAGE, 85
 - SPE_EVENT_IN_MBOX, 85
 - SPE_EVENTS_ENABLE, 86
 - SPE_EXIT, 86
 - SPE_INVALID_DMA, 86
 - SPE_ISOLATE, 86
 - SPE_ISOLATE_EMULATE, 86
 - SPE_ISOLATION_ERROR, 86
 - SPE_MAP_PS, 86
 - SPE_NO_CALLBACKS, 87
 - SPE_RUN_USER_REGS, 87
 - SPE_RUNTIME_ERROR, 87
 - SPE_RUNTIME_FATAL, 87
 - SPE_SPU_HALT, 87
 - SPE_TAG_ALL, 88
 - SPE_TAG_ANY, 88
 - SPE_TAG_IMMEDIATE, 88
 - spe_context_ptr_t, 88
 - spe_event_data_t, 88
 - spe_event_handler_ptr_t, 88
 - spe_event_handler_t, 88
 - spe_event_unit_t, 89
 - spe_gang_context_ptr_t, 89
 - spe_program_handle_t, 89
 - spe_stop_info_t, 89
- libspe2.h, 89
 - spe_callback_handler_deregister, 93
 - spe_callback_handler_query, 93
 - spe_callback_handler_register, 93
 - spe_context_create, 93
 - spe_context_create_affinity, 93
 - spe_context_destroy, 93
 - spe_context_run, 93
 - spe_cpu_info_get, 93
 - spe_event_handler_create, 93
 - spe_event_handler_deregister, 93
 - spe_event_handler_destroy, 93
 - spe_event_handler_register, 93
 - spe_event_wait, 93
 - spe_gang_context_create, 93
 - spe_gang_context_destroy, 93
 - spe_image_close, 93
 - spe_image_open, 93
 - spe_in_mbox_status, 93
 - spe_in_mbox_write, 93
 - spe_ls_area_get, 93
 - spe_ls_size_get, 93
 - spe_mfcio_get, 93
 - spe_mfcio_getb, 93
 - spe_mfcio_getf, 93
 - spe_mfcio_put, 93
 - spe_mfcio_putb, 93
 - spe_mfcio_putf, 93
 - spe_mfcio_tag_status_read, 93
 - spe_mssync_start, 93
 - spe_mssync_status, 93

- spe_out_intr_mbox_read, 93
- spe_out_intr_mbox_status, 93
- spe_out_mbox_read, 93
- spe_out_mbox_status, 93
- spe_program_load, 93
- spe_ps_area_get, 93
- spe_signal_write, 93
- spe_stop_info_read, 93
- load.c, 93
 - _base_spe_emulated_loader_present, 94
 - _base_spe_program_load, 95
 - _base_spe_program_load_complete, 96
- loaded_program
 - spe_context_base_priv, 14
- lock
 - spe_context_event_priv, 17
- lsa
 - mfc_command_parameter_area, 8
- MFC_CMD_GET
 - dma.h, 57
- MFC_CMD_GETB
 - dma.h, 57
- MFC_CMD_GETF
 - dma.h, 57
- MFC_CMD_PUT
 - dma.h, 57
- MFC_CMD_PUTB
 - dma.h, 57
- MFC_CMD_PUTF
 - dma.h, 57
- MAX_CALLNUM
 - lib_builtin.h, 81
- MFC_CMDStatus
 - spe_mfc_command_area, 25
- MFC_ClassID_CMD
 - spe_mfc_command_area, 25
- MFC_EAH
 - spe_mfc_command_area, 25
- MFC_EAL
 - spe_mfc_command_area, 25
- MFC_LSA
 - spe_mfc_command_area, 25
- MFC_MSSync
 - spe_mssync_area, 26
- MFC_OFFSET
 - spebase.h, 124
- MFC_QStatus
 - spe_mfc_command_area, 25
- MFC_SIZE
 - spebase.h, 124
- MFC_Size_Tag
 - spe_mfc_command_area, 25
- MSS_SIZE
 - spebase.h, 125
- MSSYNC_OFFSET
 - spebase.h, 125
- map_size
 - image_handle, 7
- mbox.c, 96
 - _base_spe_in_mbox_status, 97
 - _base_spe_in_mbox_write, 98
 - _base_spe_out_intr_mbox_read, 100
 - _base_spe_out_intr_mbox_status, 101
 - _base_spe_out_mbox_read, 101
 - _base_spe_out_mbox_status, 102
 - _base_spe_signal_write, 103
- mbox.h, 105
- mem_mmap_base
 - spe_context_base_priv, 14
- mfc_cmd
 - dma.h, 57
- mfc_command_parameter_area, 8
 - class, 8
 - cmd, 8
 - ea, 8
 - lsa, 8
 - pad, 8
 - size, 9
 - tag, 9
- mfc_mmap_base
 - spe_context_base_priv, 14
- mode
 - fd_attr, 6
- mssync_mmap_base
 - spe_context_base_priv, 14
- NUM_MBOX_FDS
 - spebase.h, 126
- name
 - fd_attr, 6
- npc
 - spe_context_info, 18
- PSMAP_SIZE
 - spebase.h, 125
- PUT_LS_RC
 - handler_utils.h, 68
- pad
 - mfc_command_parameter_area, 8
- prev
 - spe_context_info, 18
- Prxy_QueryMask
 - spe_mfc_command_area, 25
- Prxy_QueryType
 - spe_mfc_command_area, 25
- Prxy_TagStatus
 - spe_mfc_command_area, 25

- ps_area
 - libspe2-types.h, 89
- psmap_mmap_base
 - spe_context_base_priv, 14
- ptr
 - spe_event_data, 19
- RESERVED
 - lib_builtin.h, 81
- reserved_0_3
 - spe_mfc_command_area, 25
 - spe_spu_control_area, 31
- reserved_0_B
 - spe_sig_notify_1_area, 29
 - spe_sig_notify_2_area, 30
- reserved_108_203
 - spe_mfc_command_area, 25
- reserved_10_13
 - spe_spu_control_area, 31
- reserved_18_103
 - spe_mfc_command_area, 25
- reserved_18_1B
 - spe_spu_control_area, 31
- reserved_208_21B
 - spe_mfc_command_area, 25
- reserved_20_23
 - spe_spu_control_area, 31
- reserved_220_22B
 - spe_mfc_command_area, 26
- reserved_28_33
 - spe_spu_control_area, 31
- reserved_8_B
 - spe_spu_control_area, 31
- result
 - spe_stop_info, 33
- run.c, 106
 - _spe_current_active_context, 111
 - _base_spe_context_run, 107
 - GNU_SOURCE, 107
- SPE_CONTROL_AREA
 - libspe2-types.h, 89
- SPE_MFC_COMMAND_AREA
 - libspe2-types.h, 89
- SPE_MSSYNC_AREA
 - libspe2-types.h, 89
- SPE_SIG_NOTIFY_1_AREA
 - libspe2-types.h, 89
- SPE_SIG_NOTIFY_2_AREA
 - libspe2-types.h, 89
- SIGNAL1_OFFSET
 - spebase.h, 125
- SIGNAL2_OFFSET
 - spebase.h, 125
- SIGNAL_SIZE
 - spebase.h, 125
- SIGSPE
 - libspe2-types.h, 84
- SPE_AFFINITY_MEMORY
 - libspe2-types.h, 84
- SPE_CALLBACK_ERROR
 - libspe2-types.h, 84
- SPE_CALLBACK_NEW
 - libspe2-types.h, 84
- SPE_CALLBACK_UPDATE
 - libspe2-types.h, 84
- SPE_DEFAULT_ENTRY
 - libspe2-types.h, 85
- SPE_DMA_ALIGNMENT
 - libspe2-types.h, 85
- SPE_DMA_STORAGE
 - libspe2-types.h, 85
- SPE_EVENT_IN_MBOX
 - libspe2-types.h, 85
- SPE_EVENTS_ENABLE
 - libspe2-types.h, 86
- SPE_EXIT
 - libspe2-types.h, 86
- SPE_INVALID_DMA
 - libspe2-types.h, 86
- SPE_ISOLATE
 - libspe2-types.h, 86
- SPE_ISOLATE_EMULATE
 - libspe2-types.h, 86
- SPE_ISOLATION_ERROR
 - libspe2-types.h, 86
- SPE_LDR_PARAMS_start
 - elf_loader.h, 63
- SPE_LDR_PROG_start
 - elf_loader.h, 63
- SPE_MAP_PS
 - libspe2-types.h, 86
- SPE_NO_CALLBACKS
 - libspe2-types.h, 87
- SPE_RUN_USER_REGS
 - libspe2-types.h, 87
- SPE_RUNTIME_ERROR
 - libspe2-types.h, 87
- SPE_RUNTIME_FATAL
 - libspe2-types.h, 87
- SPE_SPU_HALT
 - libspe2-types.h, 87
- SPE_TAG_ALL
 - libspe2-types.h, 88
- SPE_TAG_ANY
 - libspe2-types.h, 88
- SPE_TAG_IMMEDIATE
 - libspe2-types.h, 88

- SPU_In_Mbox
 - spe_spu_control_area, 31
- SPU_Mbox_Stat
 - spe_spu_control_area, 31
- SPU_NPC
 - spe_spu_control_area, 31
- SPU_Out_Mbox
 - spe_spu_control_area, 31
- SPU_RunCntl
 - spe_spu_control_area, 31
- SPU_Sig_Notify_1
 - spe_sig_notify_1_area, 29
- SPU_Sig_Notify_2
 - spe_sig_notify_2_area, 30
- SPU_Status
 - spe_spu_control_area, 31
- signal1_mmap_base
 - spe_context_base_priv, 14
- signal2_mmap_base
 - spe_context_base_priv, 14
- size
 - mfc_command_parameter_area, 9
- slot
 - spe_reg128, 28
- spe
 - spe_event_unit, 21
- spe_callback_error
 - spe_stop_info, 33
- spe_callback_handler_deregister
 - libspe2.h, 93
- spe_callback_handler_query
 - libspe2.h, 93
- spe_callback_handler_register
 - libspe2.h, 93
- spe_context, 9
 - base_private, 11
 - event_private, 11
 - handle, 11
- spe_context_base_priv, 11
 - active_tagmask, 13
 - cntl_mmap_base, 13
 - emulated_entry, 13
 - entry, 13
 - ev_pipe, 13
 - fd_grp_dir, 13
 - fd_lock, 13
 - fd_spe_dir, 13
 - flags, 13
 - loaded_program, 14
 - mem_mmap_base, 14
 - mfc_mmap_base, 14
 - mssync_mmap_base, 14
 - psmap_mmap_base, 14
 - signal1_mmap_base, 14
 - signal2_mmap_base, 14
 - spe_fds_array, 14
 - spe_fds_refcount, 14
- spe_context_create
 - libspe2.h, 93
- spe_context_create_affinity
 - libspe2.h, 93
- spe_context_destroy
 - libspe2.h, 93
- spe_context_event_priv, 15
 - events, 17
 - lock, 17
 - stop_event_pipe, 17
 - stop_event_read_lock, 17
- spe_context_event_priv_ptr_t
 - speevent.h, 159
- spe_context_event_priv_t
 - speevent.h, 159
- spe_context_info, 17
 - npc, 18
 - prev, 18
 - spe_id, 18
 - status, 18
- spe_context_ptr_t
 - libspe2-types.h, 88
- spe_context_run
 - libspe2.h, 93
- spe_cpu_info_get
 - libspe2.h, 93
- spe_event.c, 111
 - _event_spe_context_finalize, 113
 - _event_spe_context_initialize, 114
 - _event_spe_context_lock, 114
 - _event_spe_context_run, 115
 - _event_spe_context_unlock, 115
 - _event_spe_event_handler_create, 115
 - _event_spe_event_handler_deregister, 116
 - _event_spe_event_handler_destroy, 117
 - _event_spe_event_handler_register, 117
 - _event_spe_event_wait, 119
 - _event_spe_stop_info_read, 121
- spe_event_data, 18
 - ptr, 19
 - u32, 19
 - u64, 19
- spe_event_data_t
 - libspe2-types.h, 88
- spe_event_handler_create
 - libspe2.h, 93
- spe_event_handler_deregister
 - libspe2.h, 93
- spe_event_handler_destroy
 - libspe2.h, 93
- spe_event_handler_ptr_t

- libspe2-types.h, 88
- spe_event_handler_register
 - libspe2.h, 93
- spe_event_handler_t
 - libspe2-types.h, 88
- spe_event_unit, 19
 - data, 20
 - events, 21
 - spe, 21
- spe_event_unit_t
 - libspe2-types.h, 89
- spe_event_wait
 - libspe2.h, 93
- spe_exit_code
 - spe_stop_info, 33
- spe_fds_array
 - spe_context_base_priv, 14
- spe_fds_refcount
 - spe_context_base_priv, 14
- spe_gang_context, 21
 - base_private, 22
 - event_private, 22
- spe_gang_context_base_priv, 22
 - fd_gang_dir, 23
 - flags, 23
 - gangname, 23
- spe_gang_context_create
 - libspe2.h, 93
- spe_gang_context_destroy
 - libspe2.h, 93
- spe_gang_context_ptr_t
 - libspe2-types.h, 89
- spe_id
 - spe_context_info, 18
- spe_image_close
 - libspe2.h, 93
- spe_image_open
 - libspe2.h, 93
- spe_in_mbox_status
 - libspe2.h, 93
- spe_in_mbox_write
 - libspe2.h, 93
- spe_isolation_error
 - spe_stop_info, 33
- spe_ld_info, 23
 - entry, 23
- spe_ls_area_get
 - libspe2.h, 93
- spe_ls_size_get
 - libspe2.h, 93
- spe_mfc_command_area, 24
 - MFC_CMDStatus, 25
 - MFC_ClassID_CMD, 25
 - MFC_EAH, 25
 - MFC_EAL, 25
 - MFC_LSA, 25
 - MFC_QStatus, 25
 - MFC_Size_Tag, 25
 - Prxy_QueryMask, 25
 - Prxy_QueryType, 25
 - Prxy_TagStatus, 25
 - reserved_0_3, 25
 - reserved_108_203, 25
 - reserved_18_103, 25
 - reserved_208_21B, 25
 - reserved_220_22B, 26
- spe_mfc_command_area_t
 - cbea_map.h, 40
- spe_mfcio_get
 - libspe2.h, 93
- spe_mfcio_getb
 - libspe2.h, 93
- spe_mfcio_getf
 - libspe2.h, 93
- spe_mfcio_put
 - libspe2.h, 93
- spe_mfcio_putb
 - libspe2.h, 93
- spe_mfcio_putf
 - libspe2.h, 93
- spe_mfcio_tag_status_read
 - libspe2.h, 93
- spe_mssync_area, 26
 - MFC_MSSync, 26
- spe_mssync_area_t
 - cbea_map.h, 40
- spe_mssync_start
 - libspe2.h, 93
- spe_mssync_status
 - libspe2.h, 93
- spe_out_intr_mbox_read
 - libspe2.h, 93
- spe_out_intr_mbox_status
 - libspe2.h, 93
- spe_out_mbox_read
 - libspe2.h, 93
- spe_out_mbox_status
 - libspe2.h, 93
- spe_program_handle, 26
 - elf_image, 27
 - handle_size, 27
 - toe_shadow, 27
- spe_program_handle_t
 - libspe2-types.h, 89
- spe_program_load
 - libspe2.h, 93
- spe_ps_area_get
 - libspe2.h, 93

- spe_reg128, 28
 - slot, 28
- spe_runtime_error
 - spe_stop_info, 33
- spe_runtime_exception
 - spe_stop_info, 33
- spe_runtime_fatal
 - spe_stop_info, 33
- spe_sig_notify_1_area, 28
 - reserved_0_B, 29
 - SPU_Sig_Notify_1, 29
- spe_sig_notify_1_area_t
 - cbea_map.h, 40
- spe_sig_notify_2_area, 29
 - reserved_0_B, 30
 - SPU_Sig_Notify_2, 30
- spe_sig_notify_2_area_t
 - cbea_map.h, 40
- spe_signal_code
 - spe_stop_info, 33
- spe_signal_write
 - libspe2.h, 93
- spe_spu_control_area, 30
 - reserved_0_3, 31
 - reserved_10_13, 31
 - reserved_18_1B, 31
 - reserved_20_23, 31
 - reserved_28_33, 31
 - reserved_8_B, 31
 - SPU_In_Mbox, 31
 - SPU_Mbox_Stat, 31
 - SPU_NPC, 31
 - SPU_Out_Mbox, 31
 - SPU_RunCntl, 31
 - SPU_Status, 31
- spe_spu_control_area_t
 - cbea_map.h, 40
- spe_stop_info, 32
 - __reserved_ptr, 33
 - __reserved_u64, 33
 - result, 33
 - spe_callback_error, 33
 - spe_exit_code, 33
 - spe_isolation_error, 33
 - spe_runtime_error, 33
 - spe_runtime_exception, 33
 - spe_runtime_fatal, 33
 - spe_signal_code, 33
 - spu_status, 33
 - stop_reason, 34
- spe_stop_info_read
 - libspe2.h, 93
- spe_stop_info_t
 - libspe2-types.h, 89
- spebase.h
 - FD_IBOX, 126
 - FD_IBOX_NB, 126
 - FD_IBOX_STAT, 126
 - FD_MBOX, 126
 - FD_MBOX_STAT, 126
 - FD_MFC, 126
 - FD_MSS, 126
 - FD_SIG1, 126
 - FD_SIG2, 126
 - FD_WBOX, 126
 - FD_WBOX_NB, 126
 - FD_WBOX_STAT, 126
 - NUM_MBOX_FDS, 126
- spebase.h, 121
 - __PRINTF, 124
 - __base_spe_event_source_acquire, 126
 - __base_spe_event_source_release, 127
 - __base_spe_spe_dir_get, 127
 - __base_spe_stop_event_source_get, 127
 - __base_spe_stop_event_target_get, 127
 - __spe_context_update_event, 128
 - _base_spe_callback_handler_deregister, 128
 - _base_spe_callback_handler_query, 128
 - _base_spe_callback_handler_register, 128
 - _base_spe_context_create, 129
 - _base_spe_context_destroy, 132
 - _base_spe_context_lock, 133
 - _base_spe_context_run, 133
 - _base_spe_context_unlock, 137
 - _base_spe_cpu_info_get, 138
 - _base_spe_emulated_loader_present, 138
 - _base_spe_gang_context_create, 139
 - _base_spe_gang_context_destroy, 140
 - _base_spe_image_close, 140
 - _base_spe_image_open, 140
 - _base_spe_in_mbox_status, 142
 - _base_spe_in_mbox_write, 143
 - _base_spe_ls_area_get, 144
 - _base_spe_ls_size_get, 144
 - _base_spe_mfcio_get, 144
 - _base_spe_mfcio_getb, 144
 - _base_spe_mfcio_getf, 145
 - _base_spe_mfcio_put, 145
 - _base_spe_mfcio_putb, 146
 - _base_spe_mfcio_putf, 146
 - _base_spe_mfcio_tag_status_read, 147
 - _base_spe_mssync_start, 147
 - _base_spe_mssync_status, 148
 - _base_spe_out_intr_mbox_read, 149
 - _base_spe_out_intr_mbox_status, 150
 - _base_spe_out_mbox_read, 151
 - _base_spe_out_mbox_status, 152
 - _base_spe_program_load, 152

- `_base_spe_program_load_complete`, 154
- `_base_spe_ps_area_get`, 155
- `_base_spe_signal_write`, 155
- `_base_spe_stop_reason_get`, 157
- `_base_spe_stop_status_get`, 157
- `CNTL_OFFSET`, 124
- `CNTL_SIZE`, 124
- `DEBUG_PRINTF`, 124
- `fd_name`, 126
- `LS_SIZE`, 124
- `MFC_OFFSET`, 124
- `MFC_SIZE`, 124
- `MSS_SIZE`, 125
- `MSSYNC_OFFSET`, 125
- `PSMAP_SIZE`, 125
- `SIGNAL1_OFFSET`, 125
- `SIGNAL2_OFFSET`, 125
- `SIGNAL_SIZE`, 125
- `speevent.h`
 - `__NUM_SPE_EVENT_TYPES`, 159
 - `__SPE_EVENT_IN_MBOX`, 159
 - `__SPE_EVENT_OUT_INTR_MBOX`, 159
 - `__SPE_EVENT_SPE_STOPPED`, 159
 - `__SPE_EVENT_TAG_GROUP`, 159
- `speevent.h`, 158
 - `_spe_event_types`, 159
 - `_event_spe_context_finalize`, 160
 - `_event_spe_context_initialize`, 160
 - `_event_spe_context_lock`, 160
 - `_event_spe_context_run`, 161
 - `_event_spe_context_unlock`, 161
 - `_event_spe_event_handler_create`, 161
 - `_event_spe_event_handler_deregister`, 162
 - `_event_spe_event_handler_destroy`, 163
 - `_event_spe_event_handler_register`, 164
 - `_event_spe_event_wait`, 166
 - `_event_spe_stop_info_read`, 167
 - `spe_context_event_priv_ptr_t`, 159
 - `spe_context_event_priv_t`, 159
- `speh`
 - `image_handle`, 7
- `spu_status`
 - `spe_stop_info`, 33
- `status`
 - `spe_context_info`, 18
- `stop_event_pipe`
 - `spe_context_event_priv`, 17
- `stop_event_read_lock`
 - `spe_context_event_priv`, 17
- `stop_reason`
 - `spe_stop_info`, 34
- `TAG`
 - `elf_loader.c`, 58
- `THREADS_PER_BE`
 - `info.h`, 76
- `tag`
 - `mfc_command_parameter_area`, 9
- `toe_shadow`
 - `spe_program_handle`, 27
- `u32`
 - `spe_event_data`, 19
- `u64`
 - `spe_event_data`, 19
- `ui`
 - `addr64`, 5
- `ull`
 - `addr64`, 5